Consistent Key-Based Routing in Decentralized and Reconfigurable Data Services

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen
Fakultät II
Humboldt-Universität zu Berlin

von
Herr M.Sc. Mikael Högqvist

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen
Fakultät II:
Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr. Alexander Reinefeld
2. Prof. Dr. Seif Haridi
3. Prof. Dr. Jochen Schiller

eingereicht am: 9. Februar 2012
Tag der mündlichen Prüfung: 4. Juni 2012

Abstract

Scalable key-based routing in distributed systems, where a message is forwarded towards a machine responsible for a partition in a large key space, has been used in many services such as key-value stores, content distribution networks and media streaming. This success can mainly be attributed to the simplicity of the route abstraction, a developer does not need to care about the mechanisms for membership management, load balancing or data replication. A limitation, however, is that most key-based routing solutions are best-effort, which means that only eventually consistent data access is possible.

This thesis presents a system (RECODE) with a key-based routing primitive called *routecast* which provides strong delivery semantics. More specifically, *routecast* guarantees that a message for a key is delivered in the same total order at a set of replicas. With stronger guarantees, applications such as coordination and metadata services as used in large storage systems or consistent key-value stores can use key-based routing. Additionally, RECODE aims to be both reconfigurable, to handle changes to the machines running the service and updates to the workload, and fully decentralized which means there is no single point of failure or bottleneck.

We make three main contributions in this thesis: 1) a group communication abstraction using primary/backup with leases for primary fail-over, 2) the design and algorithms of the *routecast*-primitive and, 3) mechanisms for atomic reconfiguration of a decentralized key space. Each part of the system is broken up into modules and presented with a specification and a set of algorithms. To validate the simplicity claim, we describe how to implement three different applications on top of RECODE. Finally, we evaluate RECODE in a cluster environment and show that the performance is competitive.

Keywords: distributed systems, group communication, key-based routing, consistency

Zusammenfassung

Skalierbares schlüssel-basiertes Routing in verteilten Systemen ist eine Methode zur Weiterleitung von Nachrichten zu den für die Partition verantwortlichen Maschinen. Diese Technik findet Verwendung in Key-Value Speichersystemen, Content Distribution Networks oder auch beim Media Streaming. Einer der Gründe für die Verbreitung ist die Einfachheit der Routingabstraktion, bei welcher der Entwickler sich nicht um die Details des Gruppenmanagements, Lastbalancierung oder Datenreplikation kümmern muss. Auf der anderen Seite sind die meisten schlüssel-basierten Routingverfahren optimistische Verfahren, bei denen der Datenzugriff keine strenge Konsistenz bietet.

In dieser Arbeit präsentieren wir das System RECODE mit dem schlüssel-basierten Routingabstraktion routecast, welches eine strengere Zugriffssemantik ermöglicht. Dabei garantiert routecast, dass Nachrichten eines bestimmten Schlüssels in der gleichen Reihenfolge an alle Replikate geliefert werden. Mit Hilfe dieser strengeren Garantien können auch Anwendungen wie Koordinations- oder Metadatendienste bzw. konsistente schlüssel-basierte Speichersysteme das schlüssel-basierte Routing verwenden. RECODE ist außerdem rekonfigurierbar bei Veränderungen der zur Verfügung stehenden Maschinen sowie bei Auslastungsänderung. Es ist ein komplett dezentralisiertes System und enthält damit keinen single-point of failure oder Systemengpass.

Die drei Hauptbeiträge der Arbeit sind 1) die Abstraktion der Gruppenkommunikation unter Verwendung von Primary/Backup mit Leases für ein failover des Primary, 2) die Entwicklung und die Algorithmen der routcast-Primitive, 3) Mechanismen zur atomaren Rekonfiguration des dezentralen Schlüsselraumes. Jeder Teil des Systems ist aufgeteilt in die entsprechenden Module und wird mit einer Spezifikation und den zugehörigen Algorithmen präsentiert. Um die Einfachheit unseres Ansatzes zu betonen, beschreiben wir außerdem drei verschiedene Anwendungen aufbauend auf RECODE. Abschließend zeigen wir durch die Evaluation von RECODE in einer Cluster-Umgebung die Leistungsfähigkeit.

Schlagwörter: Verteilte systeme, Gruppenkommunikation, Schlüsselbasiertes Routing, Konsistenz

Contents

Chapter 1

Introduction

During the last two decades there has been a paradigm change in computing systems. Large scale system have gone from being static with tailor-made hardware to more dynamic systems composed of off-the-shelf computers and network switches within data centers or even user contributed hardware connected over the Internet. Software services, and especially services storing and serving data, becomes significantly harder to design and implement when executing in a highly dynamic distributed environment. Examples of such services are distributed file systems, scalable key/value-stores and distributed services for membership management or to handle coordination of locks. Despite the complexity in design and implementation, there are two main advantages to make a service distributed or decentralized. First, it can be made to scale to accommodate more load than a single centralized service, and second, it can be made reliable to handle hardware failures, software bugs or other issues.

With more servers contributing to a service it becomes more fragile and the probability of server failures increases. Additionally, more usage leads to higher load variance which makes it beneficial in terms of resource usage to scale up or down the service by adding or removing servers. Therefore, to handle this type of dynamicity, a service must be adaptable, that is, change as the conditions change. For example, a failed server should be replaceable and it should be possible to move data between servers. For a service to be adaptable it must have access to mechanisms for doing *reconfiguration*. The decision to execute a reconfiguration can be performed by a system administrator or through an automated procedure. This decision is often based on operational thresholds the current load or reliability.

It is desirable that a service is both scalable and fault-tolerant while still being simple to operate and use. In a service storing data, reliability requires additional redundancy achieved through the replication of data over several servers. However, with replication reading or writing to the data becomes more difficult. For example, what if someone reads the data from one server while someone else updates the data at another server. What guarantees or consistency semantics should the service provide if data accesses occur concurrently?

In this thesis, we will present mechanisms for reconfiguration and data access in a scalable and reliable service for storing data. In the next section, we introduce the concept of partitioned and structured data services followed by a section summarizing the thesis contributions and the outline of the remaining chapters.

1.1 Partitioned Data Services

A data service provides an interface for performing operations on data. It can for example be a file system [GGL03], a coordination service [HKJR10] or a key-value store [RSSH09, DHJ+07]. The interface of these services is structured, meaning that it enables access to data associated with an *identifier* or *key* from a common namespace, the *identifier space* or *key space*. For example, in a file system, the file name is a reference used to find the blocks on a block-device storing the data. Similarly, a distributed file system maps the file name to a set of servers storing the file data. In a *partitioned service*, the identifier space is divided into non-overlapping (disjunctive) parts or *partitions*.

Partitioned and distributed data services have two main advantages over their centralized counterparts:

Scalability A service with several distributed servers can provide more capacity in terms of storage and throughput (client requests). Ideally, a scalable system should double the performance or storage capacity when doubling the system resources.

Fault-tolerance A single server is sensitive to failures leading to data loss. With many servers, the system can be more reliable through replication[1], wherein several servers store replicas of the same data.

[1]Other techniques such as erasure codes can also be used to improve reliability, but we do not address this topic further in the thesis.

However, when scaling a service and adding fault-tolerance, it becomes more difficult to manage. Servers fail and need replacements, more storage capacity is necessary, or data needs to be re-balanced since requests come in at a higher rate or with a different request pattern. These are all examples of *reconfiguration*, an explicit change to the service resources or the state used for managing the service, e.g. number of replicas or the partitions of the identifier space. Reconfiguration is done using a management interface provided by the service. Depending on the service design, this interface may only export a limited set of reconfigurable parameters, while others, such as the replica count, may be fixed at service start-up. A richer interface with less fixed parameters increases the flexibility of the service and reduces the need for scheduled down-time due to a reconfiguration.

1.1.1 Service State Management

A reconfiguration changes the state maintained by the service. In particular, a structured storage service has state on a set of identifiers associated with data, referred to as *items*, and a mapping of data to a set of servers. Essentially, this mapping decides who is responsible of handling read and write access to the data and is also used by clients to find the data location.

We distinguish between two models for the mapping, either each item is directly mapped to a server or the item is mapped to an identifier space partition which is assigned to a server. In the first approach, the mapping table contains an entry for each item and the set of servers storing the data for the item. This is very flexible since individual items can be placed on any available servers, but the mapping table increases linearly with the number of items. In the second approach, items are implicitly part of the identifier space partition which contains the item's identifier. For example, if the identifier space is integers between 1 and 100 with two partitions $[1, 50)$, $[50, 100)$, an item with id 42 is in the first partition. Each partition is mapped explicitly to a set of servers. With the grouping of items into partitions, the size of the mapping table becomes smaller since it depends on the number of items per partition. However, it also limits the flexibility of placing individual items at any set of servers.

The operations on a mapping table range from changes to the identifier space partitioning, addition and removal of data items and updates to the set of servers responsible for a data item or a partition. Simi-

lar to the architecture of the data service itself, the management of the mapping table is either centralized or decentralized. In a centralized solution, a single or a set of replicated servers are responsible for the entire mapping table. This is the most common approach due to its' simplicity [GGL03, HKJR10]. However, as any centralized service, it suffers from being a single point of failure and does not scale beyond the hardware of a single machine.

In the decentralized model, the mapping table itself is partitioned and each partition is assigned to a responsible server or set of servers. While this approach avoids a single point of failure and can scale by partitioning, it also introduces several new technical challenges. In particular, we need an additional table to find out which servers are responsible for what partition of the mapping table. A flexible service also provides mechanisms for this table to be reconfigured, i.e. change the responsibility of a partition to another server and modify the key space partitioning.

A special case of the decentralized model is when the item partitions and mapping table partitions are equal and assigned to the same set of servers. That is, the items are stored at the same server(s) responsible for the mapping table partition of the item. If we look at it from the perspective of indirections (reference lookups), this special case only requires a single lookup to find the data. Otherwise, two lookups are necessary, one to the mapping table followed by the redirect to the servers storing the data.

1.1.2 Consistency

A client request to an item or object in a structured data service can be divided into two parts: 1) lookup of the data location by using the items' identifier and 2) the execution of an operation on the data. The operation itself has an invocation (or start) and a response (finish). Concurrent access occurs when multiple processes execute overlapping or interleaving operations in terms of their invocation and response. A *consistency condition* specifies the guarantees on the values returned by a concurrently accessed object.

There are several well-known consistency models such as *linearizability* [HW90], *sequential consistency* [Lam79] and *eventual consistency* [TTP+95][2].

[2]Transactions are serializable which is similar to linearizability but for many objects. We do not consider transactions in this thesis.

In this thesis we focus on linearizability (also known as atomicity), which is strictly stronger than all other consistency conditions. Linearizability ensures that all operations on an object occurs atomically at some point between the invocation and the response and that there is a *global* sequential order of all operations. Intuitively, for an object providing a read and write interface, this means that a read request on any process always returns the last value written by any process. Sequential consistency guarantees a local sequential order at all processes. That is, two different processes can read different values after a write, but once the value has been read at the process, it will never read an older value. In eventual consistency, the effect of concurrent operations is undefined and must be resolved by a conflict resolution mechanism. This might lead to a single process reading a value which becomes discarded, something that cannot happen in sequential consistency.

When replicating data for fault-tolerance, maintaining a consistent system (linearizable) has several inherent costs. First, each operation require coordination which may include multiple protocol steps resulting in increased latency. Second, Brewer's Conjecture[3], later proved by Gilbert et. al [GL02], states that only two properties from Consistency (C), Availability (A) and Partition Tolerance (P) can be satisfied when implementing a replicated object. Sacrificing Consistency for Availability, resulting in an AP-system, introduces additional complexity for the developer since the same data item can be stored with different values at different servers. Additionally, sacrificing Consistency may influence the user experience. For example, when modifying a user profile in an AP-system, the change may not occur on all servers until minutes later. If the user reloads the profile page, it may see the old state and try to perform the change again. In a CP-system the profile change will be reflected immediately on at least a pre-defined quorum of the servers. However, if the required quorum does not answer the request, the service is unavailable.

1.1.3 Systems Overview

To summarize the background on decentralized data services, we non-exhaustively classify a number of different storage systems based on their architecture, consistency conditions and ability to be reconfigured at runtime. Figure 1.1 shows a venn diagram with these three characteristics

[3] Also known as the CAP-theorem

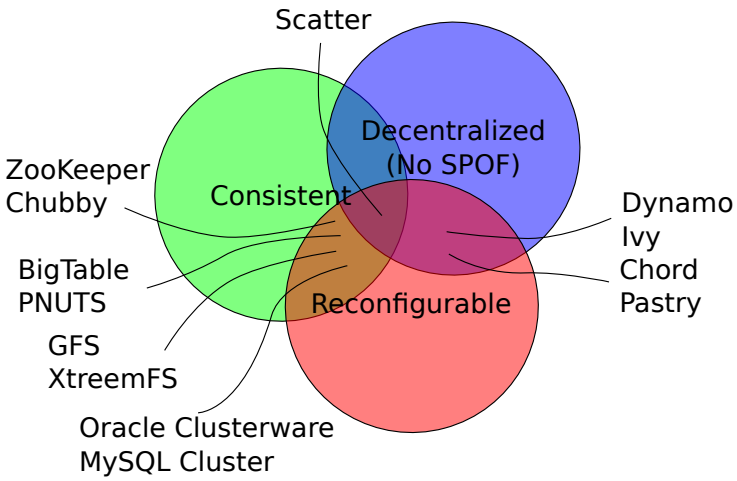


Figure 1.1: Classification of different systems depending on if they are decentralized, consistent or reconfigurable.

and a categorization of the example systems:

Consistent Indicates that a system supports strong consistency (linearizable or serializable) access to the stored objects.

Decentralized A system which does not contain any centralized components. Thus, both the data and the mapping tables for finding data are distributed to avoid any single point of failure.

Reconfigurable The system is reconfigurable if both servers and the name space can be modified at run-time.

Coordination Services. ZooKeeper [HKJR10] and Chubby [Bur06] are both examples of coordination services. Such services are an important part of many scalable distributed services. For example, distributed storage systems such as file systems [HCK+08, WBM+06], key/value stores [CDG+08, CRS+08, RST11], partition and lease management [ADW10], and partitioned transactional databases [JAM10] commonly use a centralized and fault-tolerant coordination service for serialization of requests, name lookups, replica set membership or to manage leases and locks [HKJR10, Bur06].

Coordination services must provide strongly consistent operations (linearizable or serializable) on objects. Additionally, they are often reconfigurable in the sense that servers can be replaced at run-time. To achieve

strong consistency, the operations are totally ordered and executed atomically on the service state. The total order is achieved using Paxos, e.g. in Chubby [Bur06], or Primary/Backup, e.g. in ZooKeeper [HKJR10] and the XtreemFS metadata service [HCK+08]. This enables complex operations such as read-modify-write involving single or several items stored by the service. However, the coordination required to execute these operations over the set of replicas used for fault-tolerance are also limiting throughput and increases latency for higher request rates. A single centralized coordination service may therefore become a bottleneck as the number of requests and the size of a system increases [MQ09].

Distributed Key/Value-stores. A key/value-store has a flat name space where each object is a key associated with a value. From figure 1.1, BigTable [CDG+08], PNUTS [CRS+08], Scatter [GBKA11] and Dynamo [DHJ+07] are all examples of key/value-stores. Dynamo and Scatter are both having fully decentralized name space management. Scatter provides linearizable access to objects while Dynamo has eventually consistent objects. BigTable and PNUTS are both consistent, but they use a centralized coordination service to keep track of the name space partitions which reduces scalability and is a single point of failure. In practice, scalability issues are often improved by extensive caching of the partition state either on the client-side or on dedicated machines. All systems lets the operator replace servers at run-time. Dynamo, unlike the other systems, does not provide mechanisms for re-partitioning of the name space[4].

Distributed File Systems. File systems are having a hierarchical name space with directories containing files whose data is stored in blocks or objects. Distributed file systems are typically providing consistent access to the file data since it is required by the POSIX-standard. The mapping from files to storage servers are maintained centrally in a reliable meta-data service.

A block-based file system such as GFS [GGL03], keeps track of the mapping from file to blocks (block-storage) and from blocks to storage nodes. The meta-data service often becomes a bottleneck when the number of blocks increases or when many small files are stored and ac-

[4]Strategy 3 described in Section 6.1 [DHJ+07], which has the best result on load balancing, divides the name space into Q fixed partitions, where Q cannot be changed after the system has started.

cessed in the system [MQ09]. In an object-based file system such as XtreemFS [HCK+08], the load on the meta-data service is reduced by only storing a mapping between the file-name and the set of servers storing the data for the file. The client then uses a deterministic function to calculate the location of the data.

RDBMS. Relational databases are organized into tables with rows and columns. The tables can be partitioned horizontally, e.g. row 1-100 in one partition and 101-200 in another, or vertically with columns grouped together. The operations in an RDBMS are often very complex including multi-row read/write transactions or including joins between entries in different partitions. Efficient partitioning is an active area of research [CZJM10]. The partitioning is maintained by a central coordination service and is rarely changed although individual machines can be replaced on failure. An RDBMS provides consistent access to data through transactions. Examples of shared-nothing or cluster RDBMS systems are Oracle's Clusterware[5] and MySQL cluster[6].

Distributed Hash Tables. A Distributed Hash Table (DHT) is a data service providing a put(*key*,*value*)-primitive for inserting data and a get(*key*)-primitive for retrieving data. Keys are hashed into a large integer namespace, $[0, 2^k)$, where k is usually 128 or larger. Nodes can join and leave the system and are each assigned a unique node id from the namespace. Each node is *responsible* for a range of keys which is decided based on the node's id. For example, in Chord [SMK+01], the responsibility of a node a is the set of keys in the range from it's id, a_{id}, to, but not including, it's predecessor b's id, $(b_{id}, a_{id}]$. A key is assigned to the node succeeding it in the name space. The name space wraps around at 2^k and 0 to form a ring. This method of assigning keys to nodes is also referred to as consistent hashing [KLL+97]. Pastry [RD01] is another DHT where keys and node IDs are represented as a bit-string. Keys are assigned to the node which it shares the longest common bit prefix with.

A node maintains the range of responsibility by relying on pointers to its immediate neighbors in the namespace. That is, the closest preceding node, the *predecessor*, and the closest succeeding node, the *successor*. These pointers are also used for routing. That is, the process of finding the node

[5] http://www.oracle.com/technetwork/database/clusterware/overview/index.html
[6] http://www.mysql.com/products/cluster/

responsible for a key. Using only successors/predecessors results in a linear routing cost, i.e. $O(N)$-hops, where N is the number of nodes in the system. To perform routing more efficiently, each node typically has a set of pointers to other nodes in the system structured such that routing only requires $O(\log N)$-hops. However, these pointers are not important for the correctness of the responsibility. A *routed message* is a key and a message and is forwarded greedily over the pointers by choosing the one which reduces the distance to the target the most. When a routed message arrives at the node responsible for the key (or its predecessor), the routing terminates either by an application receiving the message or the client being notified of the responsible node.

When a node joins or leaves the system the responsibility of the successor changes. To handle these changes, each node executes a maintenance algorithm periodically or react to a detected node crash by using the results of a failure detector. However, failure detectors are not always correct since it is impossible to detect the difference between a crash, a slow node or a transient failure. Most DHTs are therefore only able to provide eventually consistent ranges [KCSS07, HHMD05, SSM+08].

1.2 Contributions

Decentralized storage systems are deployed in dynamic environments such as data centers and are required to support a wide range of applications. The architecture of the system must be flexible enough to add and remove servers while, at the same time, being able to change the location of data and how the name space is partitioned. Additionally, from an operations point of view, we want to reduce possible performance bottlenecks and single point of failures. Finally, from the perspective of a developer using the storage service, strongly consistent data access such as linearizability is intuitively the easiest concept to understand and use. There are few systems which are able to provide all three of these properties at once: reconfiguration, consistency and full decentralization. In the next chapters, we will present Recode, which by using methods from reliable distributed systems in combination with the fully decentralized solutions used in DHTs, is run-time *RE*configurable, *CO*nsistent and fully *DE*centralized. In summary, the contributions of this thesis are the following:

System Model In a DHT the responsibility of a partition is decided based

on the system membership. This leads to a situation where two servers think they are responsible for the same or an overlapping partition. With a centralized service deciding on partition assignment this cannot happen. However, this does not scale and reduces system availability. We present a model where the membership is decoupled from the partition assignment similar to the centralized solution. This makes it possible to circumvent the problem of consistent partition assignment and revocation without sacrificing full decentralization.

Specification and Implementation We take a holistic view on both specification and implementation with the goal of presenting the interface, properties and algorithms necessary to implement the system modules. The structure is similar to the book Reliable Distributed Programming [GR06] from Guerraoui et. al. The design builds on well-defined modules where the implementations are replaceable as long as the specification is not violated. This modularization enables a system implementer to use the module implementation that fits best for the environment. For example, the routing method used for finding partitions can either contain a complete view of the system using broadcast or a gossip protocol or it can have partial views as in a DHT.

Partition Management Based on the our model, we present a novel mechanism to change partition assignment. This mechanism is necessary to elastically grow and shrink the resources available to the system. In order to be consistent, a re-assignment or *handover* must be atomic. We present an algorithm for the handover and argue for its correctness.

The *routecast*-primitive Key-based routing is the process of forwarding a message towards the process responsible for a partitioning containing the given key. By using the method of consistent partition management, we can introduce a new primitive, called *routecast*, with stronger guarantees. In particular, we claim that *routecast* is able to consistently and reliably deliver messages to a replicated object.

Evaluation Finally, we evaluate *routecast* and Recode both analytically and experimentally. The analytical evaluation shows that our han-

dover method is at least a factor two better than the competition. The proof-of-concept experiments show that RECODE is both scalable and can elastically accommodate load changes at run-time by adding more resources. Finally, we describe how to implement three applications on top of the *routecast*-primitive.

1.3 Thesis Outline

This thesis has the following outline: Chapter 2 introduces the background in distributed systems, and especially reliable distributed systems, that is necessary to understand the concepts presented in Chapter 3, 4 and 5. Chapter 3 introduces the model and architecture of RECODE. Chapter 4 contains the specification and implementation of a fault-tolerant group of processes based on a total order multicast-primitive. Chapter 5 presents the specification and algorithms for partition management and the *routecast*-primitive. Chapter 6 evaluates the simplicity of using *routecast* by introducing three example applications. Chapter 7 contains an experimental evaluation of a proof-of-concept implementation and an analysis of the handover algorithm complexity. Finally, Chapter 8 concludes the thesis with a discussion and future work.

Chapter 2

Background

In this thesis we use concepts from two areas of distributed systems: reliable Group Communication Systems (GCS) and partitioned storage systems with scalable routing. Both of these areas are introduced in this background section.

2.1 Distributed System Models

A distributed system consist of a set of processes that interact via a common network substrate. Two processes communicate by passing messages over the links of the network. The system is characterized by the time steps the processes need to process a message and the time the network needs to propagate a message. This time is either bounded or unbounded. For example, if a link always propagates a message within a finite time, the link is bounded. Similarly, if a process may take infinite time to process a message, the process is unbounded.

A system where the transmission delay and the processing of a message may take unbounded time is called *asynchronous*. In contrast, if there exist an upper bound on the relative speed of processes and on the transmission delay, the system is *synchronous*. The synchronous model is too strict since in real systems the network latency is not bounded and there is no bound on the relative speed of processes. On the other hand, the asynchronous model does not make any assumptions on the transmission delay or processing times. In fact, it has been shown to be impossible to deterministically solve problems such as consensus or leader election in the asynchronous model when even a single process is allowed to crash [FLP85]. The main reason is that a process cannot discern if the

network or another process is just arbitrary slow or has crashed.

Even though an asynchronous model is not restrictive enough to construct reliable distributed system, it is still possible to circumvent this issue in practice by augmenting the model with a notion of time [AW09]. Lamport showed in "The Part-Time Parliament" [Lam98], that simple timeouts are sufficient to solve consensus. In [CF99], Cristian and Fetzer introduced the *timed asynchronous* model where each process have access to a local hardware clock, which are enough to generate timeouts. Intuitively, timeouts allow processes to make sufficient progress to complete the algorithm by triggering a re-execution of an algorithm step instead of indefinitely waiting for a delayed message or a slow process.

Both of these solutions rely on the system being partially synchronous [Lyn96, DLS88]. That is, during certain periods of time (or after some period of time), the network or the processes are bounded. An alternative model is unreliable failure detectors, where each process has access to a list of other processes which are suspected to have failed [CT96] [1]. The list of suspected processes is used to avoid waiting indefinitely on messages from such a process in a protocol step, thereby allowing the executing algorithm to make progress. The algorithms presented in this thesis relies on a model where the consensus problem is solvable. We describe the exact requirements together with the implementations.

2.1.1 Processes

A process performs local computations based on an input message and produces one or more output messages. The input is stored in an input buffer and is processed one-by-one. Similarly, the generated output is stored in an output buffer. The network, through which processes communicate, provide two primitives *send* and *receive* which are adding messages to the output buffer and removing from the input buffer respectively. Local messages such as timeout events may also be added to the input buffer. The local computation for each message may take arbitrary long time (asynchronous). Each process has access to a local hardware clock and to both *volatile* and *stable* storage. The clock increases monotonically and may *drift*, i.e. increase the clock faster or slower relative to a global time. We assume that the clock drift rate has an upper bound, e.g.

[1] Failure detectors can be implemented using the partial synchrony model and how the models relate is detailed in Sec. 9.1 from [CT96]

the clock will not drift more than 1 second in 24 hours.

Failure Models. Failure models for a distributed algorithm refer to a single execution of the algorithm. A process can be in two different states: active or inactive. An *active* process executes the algorithm according to the specification while an *inactive* process does not execute anything, i.e. it has stopped.

Processes may crash due to, for example, software or hardware bugs or an administrator initiated shutdown. After a crash, a process is entering the inactive state and may from there remain inactive forever or try to recover (re-enter the active state). These two alternatives are referred to as *crash-stop* and *crash-recovery*.

In the crash-stop model, a process is either *correct* or *faulty* in terms of the execution of the algorithm. When a process is correct, it is active forever or until the algorithm has terminated[2] (which is indefinite due to the asynchronous system assumption). Similarly, a faulty process is inactive and remains inactive forever.

In the crash-recovery model, a process is either *correct*, *faulty* or *unstable*. An unstable process may crash and recover, i.e. transit from active to inactive and back to active, infinitely many times. The definitions for correct and faulty are slightly different. A correct process is eventually active forever and a faulty process is eventually inactive forever. In order to recover, algorithms mainly rely on stable storage which is assumed to be available even after a crash. Any volatile state is lost when a process crashes.

A *byzantine* process can show arbitrary behavior. For example, it may corrupt the content of a message in the output buffer or the state in stable storage. We do not consider byzantine failures in this thesis.

2.1.2 Channels

A channel is an abstraction on top of the links of connecting machines into a network. The channel is used to communicate messages between a pair of processes, i.e. to implement *send* and *receive*. A process p uses the primitive $send_{p,q}(m)$ to send a message m to q which invokes $receive_{q,p}(m)$ to receive a message m from p. Channels and links are described by

[2]Termination depends on the type of algorithm, not all distributed algorithms are able to terminate [ACT99].

the properties they provide. A common assumption about links is that they are *fair*. That is, if a message m is sent with $send_{p,q}(m)$ infinitely often, then it is received with $receive_{q,p}(m)$ infinitely often assuming both p and q are correct. The *fair links*-property corresponds to Strong Loss Limitation from [Lyn96] and *fair-loss* links from Guerraoui et. al [GR06]. A fair link may fail by losing or deliberately dropping messages during a finite period of time. A link may also reorder or duplicate messages.

A link with weak properties, such as fair link with reordering and duplication, may be used to construct a channel between two processes with stronger properties [AAF+94]. For example, a reliable channel guarantees that no messages are lost even though the used link may lose messages. This is defined with the *no loss*-property: If a process p sends a message to a process q and q is correct, then the message is eventually received by q. A reliable channel also guarantees *integrity*, that is, if q receives a message from p, then p sent the message to q. Note that the *no loss*-property require that the message is received by q even if p is unstable or faulty.

No loss is a very strong requirement and it has been shown that weaker properties are sufficient to solve, for example, consensus [GOS98]. One such property is the *quasi no loss*-property which states that if both p and q are correct and p sends a message to q, then the message is eventually received by q [ACT99]. Additionally, channels may have a FIFO-property that guarantees that messages are received in the order they are sent: if a message m is sent before m', then m is received before m'.

2.1.3 Failure Detectors

A failure detector is an oracle that reports on the state of other processes. If the failure detector (FD) returns *faulty* for a process p than p has crashed or if it returns *correct*, p is active. A reliable failure detector (RFD) eventually returns *faulty* for a crashed process and is always accurate when reporting that a process is *faulty*. An unreliable failure detector (UFD), on the other hand, may report the wrong result. The UFD returns *suspected* for a process instead of *faulty* when it believes a process may have crashed. Interestingly, as shown in [CT96, CHT96], UFDs are sufficient to solve the problem of distributed consensus.

In [CT96] Chandra and Toueg defined different classes of UFDs using two characteristics: *accuracy* and *completeness*. An accuracy-property limits the mistakes in the list of suspected processes while a completeness-property requires that a crashed process is eventually suspected. In this

thesis we will refer to two classes of failure detectors: *perfect* and *eventually weak* [3]. A perfect failure detector (or RFD) has strong accuracy, i.e. it never reports a processes as suspected before it has crashed, and strong completeness, every correct process eventually suspects the crashed process. With an eventually weak FD a correct process is eventually (after some time) never suspected by another correct process (accuracy) and if a process crashes it is permanently suspected by some correct process in the system (completeness). The eventually weak FD is equivalent to Ω, the weakest failure detector that can solve consensus [CHT96].

2.2 Specifications and Implementations

The specification of a distributed algorithm has two parts: an interface and the safety and liveness properties guaranteed by the interface. The interface defines a set of *input messages* or operations and a set of *output messages*. For example, in Section 2.1.2 we referred to the interface of a channel as *send(m)* (input) and *receive(m)* (output). We say that a process *invokes* (or *executes*) when it inputs a message to the algorithm or when the algorithm returns the results. A process p invokes *send(m)* to send a message m and a process q invokes *receive(m)* to receive m. In between the invocation of *send* and *receive*, the algorithm executes.

The safety and liveness properties of the specification refers to what and when something should happen as a result of the algorithm input. Informally, a *safety*-property indicates the valid range of the output while a *liveness*-property defines that something will eventually happen when executing the algorithm. An algorithm that executes according to the properties of the specification is valid or correct.

To make this more concrete, we make an example with the specification of a reliable channel. More specifically, the quasi-reliable channel proposed by Aguilera et. al in [ACT99]. The interface of the quasi-reliable channel from a process p to q is

in *qr-send*(m) When p invokes *qr-send*(m) it *sends* a message m to q.

out *qr-receive*(m) When q invokes *qr-receive*(m) it *receives* a message m from p.

[3] $\mathcal{P}$ and $\Diamond\mathcal{W}$ from [CT96]

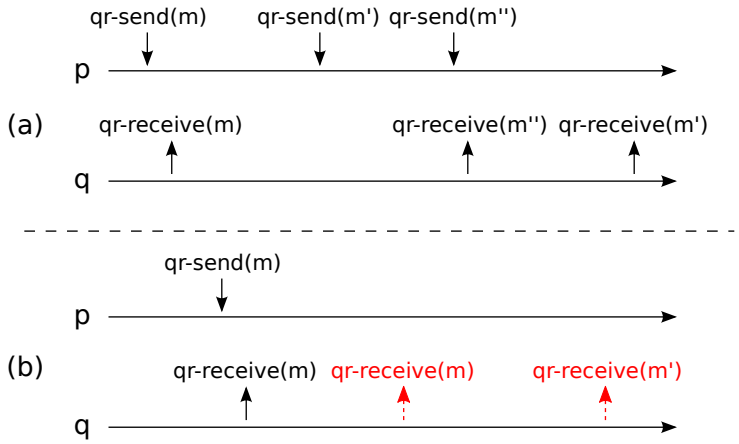


Figure 2.1: Two traces of the quasi-reliable channel specification.

The quasi-reliable channel has two safety properties and a liveness property which are defined as follows:

No Creation For any pair of processes p and q, if q executes *qr-receive*(m), then p executed *qr-send*(m).

No Duplication For any pair of processes p and q, if p executed *qr-send*(m), then q executes *qr-receive*(m) at most once.

Quasi-Reliable Let p and q be two correct processes, if p executes *qr-send*(m) to q, then q eventually executes *qr-receive*(m).

The safety-properties of the reliable channel ensures that a message is only received once by the target and that it only receives the message as long as it is sent by the sender of the message. A liveness property states that something good eventually happens. In this case, the message sent by p is eventually received by q as long as both of the processes are correct. This property is weaker than for a reliable channel where it is sufficient that only q is correct. The implementing algorithm must guarantee all of these properties.

A trace is an execution of the operations defined by the specification. Figure 2.1 contain two traces of the quasi-reliable channel specification, a) is a valid trace while b) violates both the **No Duplication**-property and **No Creation**-property. In both traces a process p sends messages to q which executes *qr-receive* when receiving the messages. Note that in the valid trace, q executes *qr-receive* for m'' before m'. This is not violating the

specification since messages do not need to be received in order, which would be the case for a FIFO channel. Trace b) violates the specification since q executes *qr-receive* for m twice and since it executes *qr-receive* for m' which was never sent by p.

2.2.1 Implementation and Syntax

In this section we introduce the style and syntax of implementations. We use an implementation of the quasi-reliable channel as an example. The syntax contains two main elements, procedures and an event-based notation for handling incoming messages of different types. All message handling executes within a single process (no concurrency). The pseudo-code contains control flow statements such as if-then-else, while-loops and repeat constructs. We introduce special key-words for sending, **send**, and replying directly to the sender of a message, **reply**. Certain primitives such as quasi-reliable send, **qr-send**, can also be used as key-words instead of procedure calls to simplify the presentation. Further, the language has sets, initialized with $\emptyset$, and maps, initialized with $\{\}$.

Algorithm 1: Implementation of send and receive for quasi-reliable channel from p to q.

```
1  received ← ∅  ▷ Buffer of received messages.
2  procedure qr-send(m) do
       ▷ Sends a QRSend()-message to q until q replies.
3      repeat periodically
4          send QRSend(m) to q
5      until receive QRSendAck(m)
6  on receive QRSend(m) do
       ▷ Check if qr-receive(m) already executed.
7      if m ∉ received then
8          received ← received ∪ m
           ▷ Executes qr-receive(m) locally at the process.
9          qr-receive(m)
       ▷ Always reply when we receive a QRSend()-message.
10     send QRSendAck(m) to p
```

Algorithm 1 presents an implementation of the quasi-reliable channel. We assume the existence of unreliable send and receive-primitives over

a fair link that can lose, delay and re-order messages. Each process has access to a local clock which is used to generate timeouts. Procedures are called by the user (in a separate thread) and may block until completed. Any code in the scope of **on receive** is executed within the process thread. Using the described syntax, the implementation is straight-forward. To send a message over the quasi-reliable channel between p and q, we call *qr-send*. The message is wrapped within the QRSend-event which is sent repeatedly from p until q replies with QRSendAck.

Correctness. We argue for the correctness of the algorithm. Since both processes are assumed to be correct and the link is fair, the message is eventually received by q. Similarly, p eventually receives the ack by re-sending the request (line 6 and 10). q maintains a set of received messages in order to avoid executing *qr-receive* more than once (line 7-9) guaranteeing **No Duplication**. Finally, since the link is not creating new messages (not byzantine) that where not sent by p and q only reacts on messages arriving through the link, q will only execute *qr-receive* for messages sent by p which guarantees **No Creation**.

2.3 Process Group Communication

Group communication systems [CKV01] were introduced to provide abstractions and primitives operating over sets of processes. We call this set a *process group*. The group provide primitives for communicating with all the processes in the group and for changing the composition of the group by adding and removing processes. A *broadcast*-primitive is used to communicate with all members of the group while a *multicast* is for a specific set of processes. The primitives provide different levels of guarantees. For example, a best-effort broadcast does not guarantee that all process delivers the broadcast message while a total order broadcast guarantees that all correct processes deliver messages in the exact same order. For group communication primitives we use *deliver* instead of receive, since a process may for example receive a message before it is allowed to deliver it. We first introduce the consensus problem, which is vital for reliable group communication primitives. This is followed by total order broadcast, also known as atomic broadcast which relies on consensus, and a brief introduction to primary/backup-based coordination and the concept of replicated state machines. Finally, we discuss issues related to

group membership.

2.3.1 The Consensus Problem

Agreeing on a value among a set of participants has turned out to be one of the most important problems in reliable distributed systems. It is the basis for many other problems such as leader election, membership management and total order delivery. A consensus protocol is also a practical abstraction for replicating the same state among a set of processes which is useful in, for example, a fault-tolerant database.

Consensus has two primitives: *propose(v)* and *decide(v)*. Any process that wants the process group to agree on a value uses *propose(v)* to introduce the value. Several process may propose values concurrently. When the consensus protocol has reached an agreement on a value, all processes execute *decide(v)*. We summarize the properties of agreement as follows [BDFG03, Lam98, CT96]:

Validity If a process decides on a value, then it was proposed by some process.

Integrity Every process decides at most once.

Agreement No two processes decide differently.

Termination Every process eventually decides.

Validity, Integrity and Agreement are all safety properties while Termination is a liveness property.

2.3.2 Total Order Broadcast

Atomic broadcast or total order broadcast is a group communication abstraction that guarantees that messages are delivered at all group members in an agreed upon order. Intuitively, if two messages, m and m' are delivered to all members, then either m is delivered before m' or m' is delivered before m. That is, if some process delivers in the order m, m', then all processes must deliver the messages in that order. An atomic/total order *multicast* is a more general primitive that targets a subset of processes of the system unlike a broadcast which addresses all processes. We use multicast when the members of the process group can change.

[DSU04] contains an excellent summary and taxonomy of existing total order broadcast specifications and implementations. They introduce two important terms for message ordering mechanisms: *destination agreement* and *fixed sequencer*. Destination agreement means that the order of messages are agreed upon among the group members using a consensus algorithm. One such algorithm is Paxos [Lam98], which we present in Section 2.3.2. In a fixed sequencer algorithm, a single dedicated process decides on the message order before it is broadcasted to the other group members. Primary/backup, described in Section 2.3.2, is a fixed sequencer algorithm where the sequencing process is called the primary. While the actual broadcast algorithm for fixed sequencer is simpler to implement than with destination agreement, it becomes more complex to add fault-tolerance when the single sequencer fails. Total order broadcast has been shown to be a useful abstraction for distributed and fault-tolerant databases [Ped99, Wie02].

A total order broadcast (TO-broadcast) service exports two primitives: *to-broadcast(m)* and *to-deliver(m)*. *to-broadcast* is executed by a process to broadcast a message m and *to-deliver* is invoked at each process in the group when the protocol has decided to deliver the message. In addition to the properties for consensus, TO-broadcast introduces a property for ordered delivery:

Total Order If two processes p and q both invoke *to-deliver* for m and m', then p invokes *to-deliver* for m before m' if and only if q invokes *to-deliver* for m before m'.

Interestingly, it was shown that the Consensus and TO-Broadcast problems are equivalently difficult to solve in any asynchronous system (they both require the Ω failure detector) [CT96, CHT96].

Paxos

Paxos was invented by Lamport and first published in "The Part-Time Parliament" [Lam98]. The algorithm solves both the problem of consensus (a single Paxos or consensus instance) and total order broadcast (by chaining instances together). In that sense, Paxos satisfies the properties of both consensus and TO-broadcast as specified above. Since the order of delivery is determined through an agreement (consensus), the Paxos algorithm is a destination agreement algorithm [DSU04].

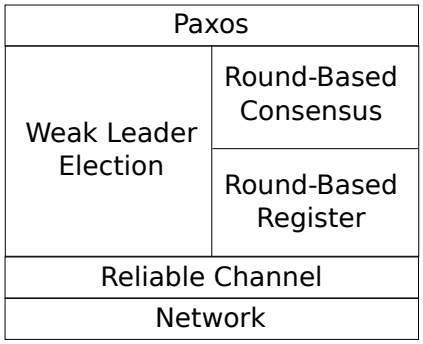


Figure 2.2: Paxos modules, (adapted from [BDFG03]).

Paxos executes in an asynchronous model where processes may crash and recover and where links are unreliable. The protocol for a single paxos instance has two phases, a proposal-phase which can be initiated by any process and a phase for accepting the proposal. Once accepted, the proposed value for an instance does not change. Multiple processes can propose values concurrently in the same instance and are distinguished by a process unique and always increasing *round*-identifier[4]. The processes agree on value when a process eventually is able to both propose and decide on a value without other concurrent proposals. Total order delivery is achieved by assigning each instance an increasing identifier and never deliver a message if there is a gap in the delivery sequence. Progress is ensured as long as a majority of processes are responding in a timely manor. This is similar to the partially synchronous model where there are periods when the link delays or processes act on messages within a bounded time.

We describe Paxos from the modularized perspective used in "Deconstructing Paxos" [BDFG03]. Figure 2.2 gives an overview of the different modules from the deconstructed Paxos. Each process running the protocol executes each module in a separate task or thread[5]. The network link may drop messages and a *Reliable Channel* module is used to ensure similar guarantees as the quasi-reliable channel described in sec. 2.1.2. The remaining modules are described bottom-up.

Round-Based Register. The round-based register is a shared, majority-based register for reading and writing a value. This module exports two

[4]Also known as the ballot number [Lam01]

[5]In [Lam01], there are different roles for a process: *proposer*, *acceptor* and *learner*

primitives: *read(k)* and *write(k,v)*, where k represents a round and v a value. A read returns a (*outcome*, v)-tuple where the outcome is either commit or abort, while the write only returns the *outcome*. If the register is empty (i.e. no committed write), it contains the value $\perp$. The properties of a register are derived from the propose and accept-phases of a single Paxos instance. Specifically, the properties for a round-based register are [BDFG03]:

Read-Abort If $read(k)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' \geq k$.

Write-Abort If $write(k, *)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' > k$.

Read-Write-Commit If $read(k)$ or $write(k, *)$ commits, then no subsequent $read(k')$ can commit with $k' \leq k$ and no subsequent $write(k'', *)$ can commit with $k'' < k$.

Read-Commit If $read(k)$ commits with v and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' < k$.

Write-Commit If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.

The round-based register ensures that if there was a successful write, a read always returns the last value written. Two processes competing to read or write the register may never commit if they increase the k and the operations overlap. In practice, when local time is available, this can easily be solved by random timeout before sending the next request.

Round-Based Consensus. The round-based consensus module represents a single consensus instance. It exports the primitives *propose(k,v)* and *decide(v)*, where k is the current round and v the proposed value. Any process can propose in a consensus instance, but the round, k, must be unique and a process is not allowed to propose in the same or in a lower round as a committed proposal. The round-based consensus relies on the round-based register. The two phases, propose and accept, are implemented using a read followed by a write of the register. If both the read and write committed, it means that no other processes tried to propose concurrently and that the consensus value is *stable*. A stable value

is returned by any subsequent read of the register. If a process tries to write, it must always read first ensuring that the value is never changed.

Weak Leader Election. The weak leader election eventually elects a common leader among a set of processes. Each process has access to a local *leader*-operation which returns the process identifier of the process that is currently believed to be the leader. The leader election is *weak*, since *leader* may return different leaders at different processes at the same global time. However, eventually all processes returns the same leader. Multiple leaders does not violate the safety-properties of consensus but may lead to problems with liveness if none of the processes are able to read and write to the round-based register. A solution to this is to force random or exponentially increasing timeouts between proposing a value.

Paxos and Total Order Delivery. The paxos module ensures that messages are delivered according to a total order. Essentially, it defines how to chain multiple consensus instances and how to recover an unstable process (one which repeatedly crashes and recovers). An application with requirements on total order delivery is built on top by using the exported *to-broadcast(m)* and *to-deliver(m)* primitives. This module depends directly on the round-based consensus and weak leader election modules.

Primary/Backup

In a Primary/Backup protocol [BMST93], a single dedicated processes (primary) sequences all broadcasts according to a total order. The other processes are backups, that are ready to replace the primary when it fails. This coordination scheme is referred to as single sequencer [DSU04]. Unlike Paxos, there may only be a single primary at any point in time, i.e. the leader election is *strong*. With a strongly elected primary, both the consensus and total order delivery is solved deterministically at the primary. However, for the agreement and ordering decision to be *stable* the primary must first make sure that the backups can recover the decision in case of a primary failure. A message is stable when a sufficient number of backups (a majority or pre-defined quorum) have received and acknowledged the message to the primary. Chapter 4 contains an implementation of a primary/backup-based process group which provides primitives for total order delivery.

Replicated State Machines

A replicated state machine (RSM) deterministically executes operations in the same order at a set of replicated processes [Sch90]. Since the operations are all applied in the exact same order on all processes the set of processes can be perceived as a single fault-tolerant process. An RSM can be specified using the specification for total order broadcast as long as the delivery sequence is gap-free [CKV01]. The gap-free property for (uniform) total order is defined as follows [DSU04]:

Gap-Free Uniform Total Order If some process delivers a message m' after message m, then a process delivers m' only after it has delivered m.

Each process that delivers a message applies it to some local state. The execution of the message must be deterministic, since otherwise the processes could end up in different states.

2.3.3 Group Membership

So far, we have only considered process groups where the set of members are fixed. A fixed or *static* group does not allow the group members to change over time. In a static group with crash-stop processes, a process cannot recover after a failure nor can a new process be added to the group. Even though the static group with crash-stop processes model is often used when theoretically describing distributed algorithms such as Paxos, the model is not very useful in practice. An alternative is to assume that processes are crash-recovery or to use the *dynamic* group model where the group may change the set of members over time. Implementing a dynamic group and processes that are crash-recovery requires additional algorithms for handling failures and the membership of the group.

Failure of processes stop an algorithm from making progress, i.e. it may violate the liveness properties. By using a non-static crash-recovery or a dynamic model, we can increase the probability of the group to survive, i.e. continue making progress. For example, if an algorithm requires a majority quorum for making progress, only a single process in a group with three processes may crash. Recovering a process is often a manual intervention from an administrator that restarts a hanging process or exchanges system hardware.

In the static crash-recovery case there is still a chance that a majority of processes crash during, for example, server maintenance. This can be alleviated by using a dynamic group where the fault-tolerance can be increased temporarily while replacing system hardware [CGR07]. The policy decision for changing a dynamic group can use principles from self-management, e.g. increase or decrease the size based on thresholds. We call the change of the group a *reconfiguration* or *migration* [LAB+06, LMZ10].

Views. In a dynamic group, a *view* represents the group configuration, i.e. the set of members $S \subset \Pi$, where Π is the set of all processes. Each view is associated with a monotonically increasing version, thus, the view is a tuple: $\mathbb{N} \times \mathbb{P}(\Pi)$. The view that is currently in use is *installed*. A process that is in the next installed view is *added* to the group and a process that is not in the next view is *removed*. We also say that a process *joins* or *leaves* a view. A process in a view is *correct* in that view if it does not fail and is part of the next view, otherwise the process is *faulty* in the view. Additionally, if a process is faulty in any view it is *group faulty*, otherwise the process is *group correct*.

Application State. Between the group communication layer and the application there is a clear separation. The application uses the primitives defined by the group communication specification. When a group member recover its state after a failure or reconfiguration it performs a *state transfer* from one of the other members or from the local stable storage. The application state is recovered when the group member replays the messages (notifies the application layer) from the recovery. Thus, the group communication layer has a responsibility to maintain the state necessary to recover the application.

There are two alternative approaches for recovering a crashed process in a process group, with or without stable storage. When stable storage is available, a process recovers all messages or a checkpoint [LMZ10] directly from the storage. Additionally, the process may have fallen behind the other processes and must recover the remaining state by asking them for any missing messages. Without stable storage, the process directly asks the other processes to transfer their latest checkpoints and remaining messages. None of these methods violate the safety properties [6], but

[6] A recovering process must still remember the process id.

have different characteristics in terms of fault-tolerance and resource usage. For example, if processes lack stable storage it may be sufficient with a power failure to crash the critical set of processes that are required to make progress.

Chapter 3

Recode Overview

In the background chapter, we introduced two areas of distributed computing: reliable group communication and partitioned data services. Group communication systems (GCS) are providing primitives such as reliable or total order broadcast with well-defined semantics. These primitives simplifies the development of reliable and consistent distributed services [Bir93]. However, due to the coordination required to execute an operation in the process group, the scalability is limited. That is, adding more hardware or group members does not increase the performance.

Partitioned data services provide storage and access to data associated with an identifier from a large name space. The identifiers and data are deterministically mapped to the machines contributing resources to the service. Distributed Hash Tables (DHTs) is one way to implement a partitioned data service. A DHT is fully decentralized, and thereby scalable, and reconfigurable in the sense that mechansisms exist that can modify the identifier mapping and that let machines join and leave the system at run-time. DHTs are however not able to provide consistency in the event of machine failure or a reconfiguration [Ris07, SSM+08, Gho06].

In this chapter we introduce the model and architecture of Recode, a system which uses the concepts for decentralization and reconfiguration from DHTs and the abstractions for achieving consistency and process membership mechanisms from GCSs. This lets us provide a system which is scalable, run-time reconfigurable and consistent. At the core of our approach is a primitive called *routecast*. Intuitively, routecast forwards a message to a set of processes responsible for a key in a partitioned identifier space.

An important design decision has been to separate mechanisms from

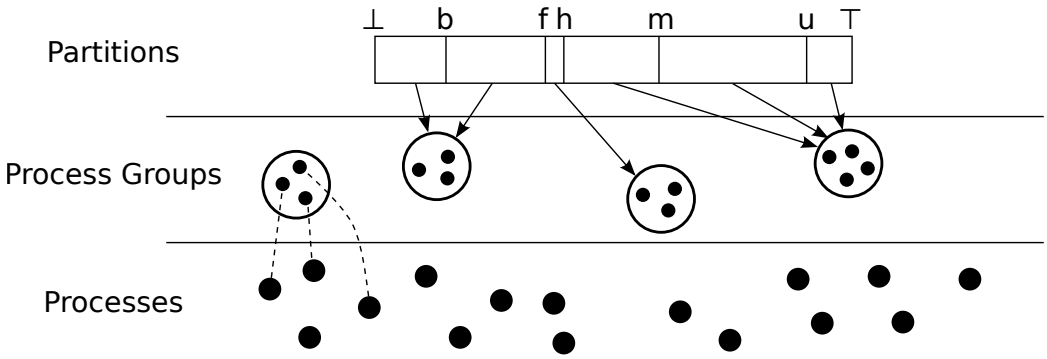


Figure 3.1: System model overview with partitions, process groups and processes.

policies. This means, for example, that we describe how to add a new process to the system, but not why we add the process. The reason to add or remove a process is a decision taken by another service. To have a self-managing system, the presented mechanisms can be used by applying techniques from autonomic systems [ASFPV10]. The importance of separating concerns for distributed algorithms has been advocated in for example [ST06] and shown practically in [Sch06].

3.1 System Model

The basic elements of the our model is a partitioned name space and processes. Sets of processes are tightly coupled into process groups. These groups use group communication primitives to mimic a single process, i.e. a reconfigurable replicated state machine. Figure 3.1 shows how these three elements interact. From the bottom up, processes are members of process groups and groups are responsible for partitions. A process may exist without being a member of a group and a group may not be responsible for a partition. However, all partitions must be assigned to some process group. If this is not guaranteed, partitions may be forgotten which results in data loss.

A significant difference in our model compared to a DHT is that process membership is decoupled from the management of the name space partitions. In a DHT, each individual process is responsible for a partition. If a process becomes slow or the network drops messages, another process automatically takes over this process' partition through the DHT

maintenance protocol. However, detecting a failure using a failure detector or through a periodic monitoring message may return an incorrect answer, e.g. a process may still answer requests from other processes without receiving the monitor request [SSM+08]. In the DHT model, this error may lead to the re-assignment of the responsibility for partition from a still correct process without it even knowing that its not responsible anymore. Thus, the system ends up in an incorrect state where two processes are responsible for an overlapping partition. Decoupling the responsibility revocation and assignment of partitions from the membership decision makes it possible to avoid this inconsistency.

Furthermore, this decoupling has three additional advantages. First, process groups may be responsible for more than one partition. This makes it possible to balance the load more fairly between groups as shown in [LS05, SMK+01]. A fair load is necessary to efficiently use the system resources. Second, each group can have different number of member processes. This can for example be useful if some partitions require higher reliability. Finally, when partitions are associated with state, the data movement between groups becomes an explicit decision instead of occurring each time a process fails, joins or leaves the system.

3.2 Architecture

In this section we give an overview of the modules that are used to implement a ReCoDe system. The upcoming chapters (4 and 5) on process groups, routecast and partition management will present a complete specification and the algorithms used to implement the different modules. The modularization simplifies the description of the specification and algorithms, but it also makes it easier to reason about their correctness. Moreover, with well-defined interfaces and a clear separation between the different modules lets us compose a system with different properties. For example, ReCoDe contains a module for routing messages towards an identifier in the partitioned name space. The cost of routing a message can be implemented to take $O(1)$ or $O(\log N)$-hops depending on the system size.

Figure 3.2 shows the modules implemented by each process in the system. An application on top of ReCoDe, uses the interface provided by the different modules to perform operations. For example, when the application routecasts a message it sends a *routecast(key, message)* input

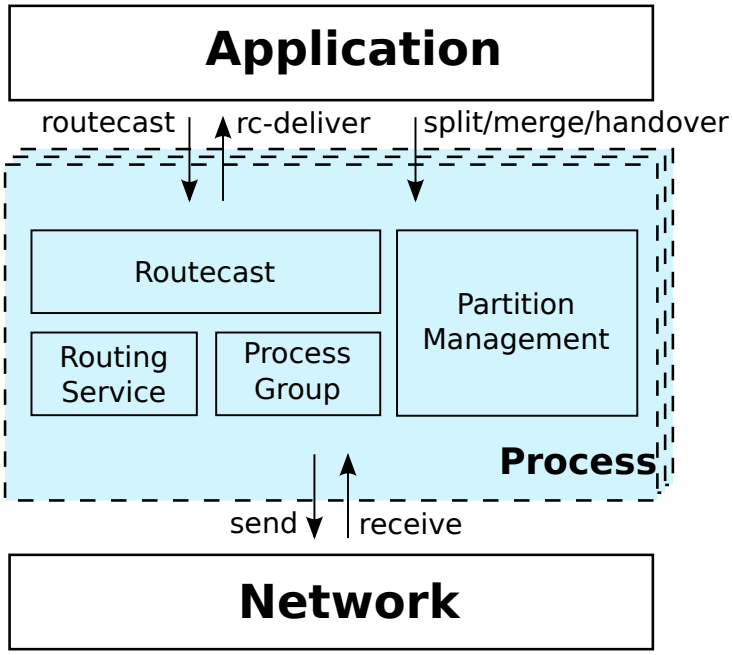


Figure 3.2: The architecture and modules of a RECODE system.

event to the routecast-module at the process. When the routecast has finished executing, it terminates by triggering an *rc-deliver(key, message)* output event at the processes responsible for the key. How to handle the rc-deliver event is application-specific. Chapter 6 presents some example applications for how to use RECODE.

Network. Each process has access to a network module which allows it to send and receive messages from other processes. In addition to unreliable message send, the network provides a quasi-reliable channel that is used by invoking *qr-send* and *qr-receive* (described in Section 2.2.1). This channel guarantees that a message is received at most once as long as both of the processes are correct.

Partition Management. The partition management module maintains the state of local partitions that a process is responsible for. It provides three operations on the state: *split*, *merge* and *handover*. A split divides a partition into two non-overlapping partitions. The merge combines two adjacent partitions and the handover changes the ownership from one process group to another. The partition state is made reliable by replicat-

ing it on the processes in a process group.

The *split* and *merge* primitives are both executed within a single process group. For a *merge* it is assumed that both partitions are stored at the group. The *handover* must be atomic and requires coordination between two process groups. Atomicity is necessary to guarantee the main property of the *routecast*-primitive: messages delivered to the same key are always delivered according to a total order. If two process groups both are responsible for a partition covering a key, they would be able to deliver messages which violate the total order. For example, let a group A deliver messages $(1, k, m)$ followed by $(2, k, m')$. In between the delivery of the messages the partition containing k is handed over to B, which subsequently delivers a message $(2, k, m'')$. $(2, k, m')$ and $(2, k, m'')$ are concurrent. However, if the handover is atomic, A will never deliver $(2, k, m')$.

Process Group. The process group provides primitives for total order multicast, initializing and destroying a group as well as adding and removing processes from the system. The total order multicast is used to implement a fault-tolerant replicated state machine (RSM) and the group management operations are all executed as operations within the RSM [Sch06, LAB+06]. This makes it possible to increase and decrease the number of group members at run-time.

The fault-tolerance of a group depends on the properties of the implementation. For example, a Paxos-based process group can handle up to f concurrent failures in a process group with $2f + 1$ members. If more than f processes are temporarily unavailable the group will not be able to make progress, i.e. execute operations. A strong assumption of a deployed ReCode system is therefore that the process group has a sufficient number of members (replication factor) to handle the failure characteristics of the underlying resources.

As surveyed in [DSU04] and [CKV01] there exist a wide range of group communication implementations with different properties. Recent research has focused on reconfiguration of replicated state machines [LAB+06, LMZ10], optimizations for Wide-Area Networks [MJM08] and high-throughput in Local Area Networks and clusters [MPP11]. We present a complete implementation of a primary/backup-based process group in Chapter 4.

Routing Service. The routing service is used to locate a process from the process group that is currently responsible for the partition covering a given key. The service uses an internal look-up table from key to partition and process group. When the system is reconfigured by moving partitions between process groups or when processes join and leave, the service is updated. Unlike the partition management, the routing service does not need to be consistent, i.e. the local state must not reflect the most current partition assignment or process membership. However, if the state is not updated the service will stop working when a route-request cannot reach the key owner. Thus, if there are no reconfigurations, we require that the routing service eventually contains the latest system state.

The routing service can be implemented as a library at the application clients, as a separate service with dedicated machines or as part of the processes in the process group and the topology can be of any type, e.g. ring, complete map or a tree. The main trade-offs are the look-up latency and the cost for updating and maintaining the state. There exist several types of routing topologies and overlay networks which can be used to implement the routing service [RS04, SMK+01, SSR08].

Routecast. The *routecast*-primitive uses the routing service to forward a message towards the responsible process group. When the message is received, the process group decides if it should be delivered or not, i.e. is the group responsible for the partition covering the key. This decision uses internal state from the partition management module. The total order primitive of the process group enforces the message delivery order. Delivered messages are handled by the application. For example, an array of atomic registers has handlers for read and write at each array entry (key). The atomicity of read and writes are guaranteed by the total order properties (linearizable).

3.3 Related Work

DHTs uses two primitives, *join* and *leave*, to 1) reconfigure the name space, 2) handle process (or node) membership and 3) manage the routing topology. The basic routing structure is a double-linked ring where each node is an entry with predecessor and successor pointers. The name space is partitioned by using the ID's of nodes. A node is responsible for

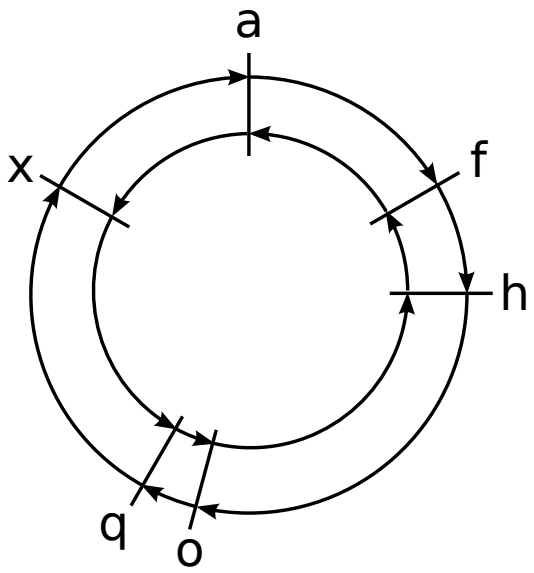


Figure 3.3: A DHT ring with six members connected by successor and predecessor pointers.

the range between its ID and that of its predecessor, i.e. a node is allowed to answer routed messages for keys between itself and it's predecessor.

Figure 3.3 shows a DHT ring with six nodes (partitions) and the predecessor and successor pointers at each node. With this configuration, f is responsible for the range $(a, f]$[1]. If a node c joins, the name space is reconfigured by splitting the range $(a, f]$ into $(a, c]$ and $(c, f]$ and the successor and predecessor pointers of a and f are changed to point at c. Because of how partitions are assigned to nodes, by using the predecessor pointer, reconfiguring the name space is the same operation as joining (or leaving) the ring. An atomic reconfiguration of the ring, requires an operation where both the joining node and it's successor agrees on updating their predecessor pointers at the same time (atomically). Thus, atomic reconfiguration of the name space is the same as consistently maintaining the ring topology.

In order to provide data access with strong consistency, the system must guarantee that each range has an exclusive owner. That is, when a process delivers a routed message to the application, it must be the exclusive owner of the range containing the key. It was shown by both Ghodsi [Gho06] and Risson [Ris07], based on the proof of Brewer's con-

[1]Including f and excluding a

jecture [GL02], that it is impossible to perform atomic changes to a ring topology without sacrificing availability while providing data consistency.

Risson [Ris07] suggest to use Paxos commit to perform consistent modifications to a ring topology. The commit participants are the predecessor, successor and the joining or leaving node. From the properties of Paxos commit, a topology change itself is non-blocking and allows for a single node to fail while still completing the operation. During the topology change, the range between the predecessor's id and the joining or leaving node's id are blocking requests. In terms of fault-tolerance, it becomes critical if a joining node fails and the protocol ends in a commit. Thus, a failed node is now responsible for a range. This failure must be detected by the successor or predecessor and a new commit must be performed to force the failed node out of the system. During this commit, no additional failures are allowed to occur. In addition, the protocol does not allow more than three members, why it is not possible to increase the reliability. Risson's solution does also not handle replication.

Both Ghodsi and Lynch et. al. [Gho06, LMR02] introduce protocols which assume that nodes are fault-tolerant. Lynch represent each DHT node as a replicated state machine and propose an algorithm for join and leave similar to the approach in Chord. Ghodsi's algorithm provides atomic change of a double-linked ring topology with fault-tolerant nodes. If nodes can fail, the algorithm becomes eventually consistent through a stabilization mechanism unless the failures result in a loopy ring [Gho06]. The algorithm uses locks to indicate when a node is taking part in a topology change. Similar to Risson's approach, during a change, the successor node cannot deliver any requests to the application for the range between the joining/leaving node's id and the successor's id.

Shafaat et. al [SSM+08] showed that incorrect failure detectors may result in nodes believing that they are responsible for overlapping partitions. This conflict is eventually resolved by the maintenance algorithm. However, in a data service, such as a DHT, even the shortest period of overlapping responsibilities may lead to two different nodes reading or writing to the same key. Thus, a DHT cannot correctly provide atomic data access, where clients must read the latest write, without significant modifications to the join and leave algorithms or the system model.

Lynch et. al [LMR02] suggested a solution to this problem by introducing a Replicated State Machine (RSM) as a fault-tolerant process. With non-failing processes, there is no need for failure detectors and it is assumed that a process eventually responds to a request. In Lynchs'

approach, the ring is maintained using a variant of Chord. However, it is not described how changes to the ring, join and leave of an RSM, is done atomically without violating data access.

Scatter [GBKA11, BGKA10] extends on the initial idea from Lynch et. al of using RSMs as nodes. They maintain the ring atomically by using a two-phase commit (2PC) which executes whenever the node itself changes the process membership, when a direct neighbor (successor or predecessor) changes or when the partitioning is updated. While it is claimed that Scatter provides provable consistency, there is no such proof nor algorithms in the articles describing Scatter. Additionally, since Scatter is maintaining a ring-structure for routing, each change of the predecessor or successor pointer or node membership results in a 2PC between three RSMs. This incurs a higher overhead compared to the partition management we propose which is either local in a single RSM or between two RSMs.

3.4 Summary

This chapter has given an informal introduction and motivation of to the design of Recode. In particular, we have introduced the concept of partitions, processes and process groups followed by the Recode modules. The next chapters will go into details of each module by presenting their specification and implementation.

Chapter 4

A Fault-Tolerant Process Group

In this chapter we describe the implementation of a fault-tolerant process group. The group provides primitives for total order multicast, operations to update the group membership and to initate and destroy the group. The presented implementation is based on a Primary/Backup-scheme. This means that a primary (or leader) is responsible for the reliability and ordering of incoming multicast requests. The backups are used to ensure fault-tolerance and are can take over the primary-role if the current primary fails. The presented algorithm is based on a fixed sequencer with fail-over according to Défago's categorisation [DSU04] of broadcast and multicast algorithms. By ensuring that the message sequence is gap-free, i.e. all processes delivers all messages in the same order, we can provide the functionality necessary to implement a replicated state machine. Essentially, each message event represents a state transition.

The total order multicast algorithm is kept simple by making the following two assumptions. First, the primary only has a single outstanding operation at a time. That is, there is no pipelining or message batching as in, for example, SMART [LAB+06]. Adding these optimizations are not restricted by the specification. Second, only a single group member is added or removed for any reconfiguration operation. This ensures that the quorum in the previous view of the group and the new view are always overlapping. Without this property, we can reach an invalid state where processes are executing multicasts in two different processes sets.

Figure 4.1 contains an overview of the different components used to implement a dynamic process group. From the top, we have the application which uses the primitive *to-multicast* for executing a total order

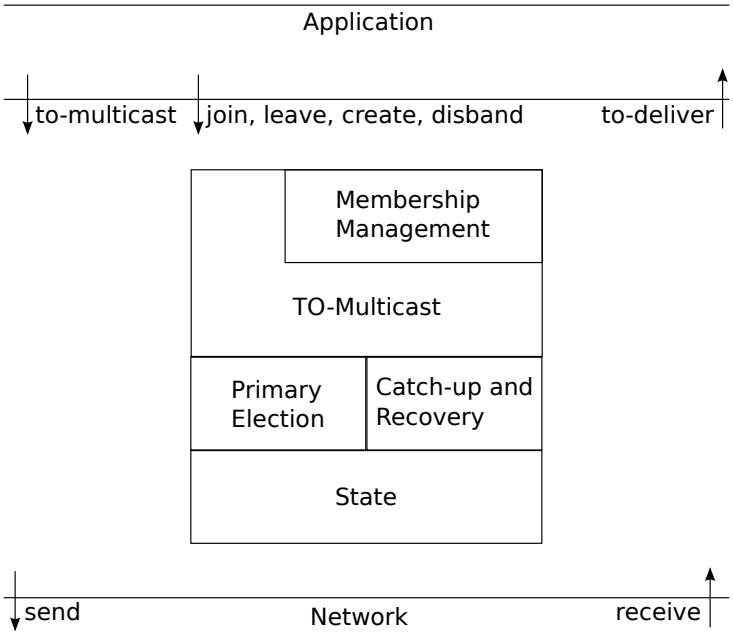


Figure 4.1: An overview of the components for TO-multicast and process group management.

multicast operation, which terminates with the invocation of *to-deliver* at the processes in the group. To modify the process group, the application can trigger a join or a leave, create a new group or destroy the group. The components implementing these primitives are 1) TO-Multicast which orders the incoming requests, 2) Membership Management for reconfiguring the group, 3) Primary Election which is used to decide on which process has the primary role, 4) Catch-up and Recovery to handle process failures and temporary message loss, and 5) the State component which holds the complete operations log.

In the section 4.1, we introduce the system model and definitions used in the rest of this chapter. This is followed by section 4.2 which more formally introduces the primitives and guarantees provided by the process group. A solution to the specification is presented in section 4.3, followed by related work, section 4.4, and finally a discussion in section 4.5.

4.1 System Model and Definitions

A distributed system consists of a finite set, Π, of processes. Processes communicate by passing messages and react on messages to access or modify locally stored state. Incoming messages are translated into operations and executed in some order at each process. A message m is from the set of valid messages $\mathcal{M}$. The sender of m is denoted $sender(m)$.

A group consist of a subset of processes from Π. Groups are dynamic, i.e. processes can join and leave. Processes in a group can fail and recover. A specific group configuration is called a *view*, and defined by a group id, a view id and a set of member processes, $(group, id, S)$. We say that a process in a view is *view correct* in that view if it does not fail and is part of the next view, otherwise the process is *view faulty*. Thus, a process can be correct in view 1 to view 5 and view faulty in view 6 since it is not part of view 7. Additionally, if a process is faulty in any view it is *group faulty*, otherwise the process is *group correct*. A group correct process is view correct from the view it joined the group until the group is destroyed. A process group that replicates state over all members, *stores* that state. We assume that any state stored by the group is transferred between views.

For the implementation[1], we are assuming a timed asynchronous system model [CF99] with links where messages may get lost, re-ordered and delayed. Each process has access to a local clock c_p which increases strictly monotonically, i.e. $c_p(t) < c_p(t')$, then $t < t'$.

The set of processes in the current and next view have loosely synchronized clocks. That is, the clocks at any pair of processes, $c_p(t)$ and $c_q(t)$, there is a known upper bound on their difference, ϵ. At any time t and view (k, S), the following condition must hold: $\forall p, q \in S : \epsilon \geq |c_p(t) - c_q(t)|$. We assume an external service such as NTP [Mil91] which maintains a correct clock. This service is accessed periodically to synchronize the process clocks in order to adjust for local clock drift. If a process cannot synchronize within the interval necessary to guarantee ϵ, it stops answering requests until it has been able to synchronize again. This does not violate the safety properties of the proposed algorithms, but may increase delays that affects the total time of an algorithm execution.

[1] To implement the specification it is sufficient with a partially synchronous system which is enough to solve consensus. We use a lease algorithm for fail-over which requires the timed asynchronous model.

4.2 Specification

In this section we introduce the interface and properties of a dynamic group with a total order multicast (TO-multicast) primitive (TO-broadcast in a static group). The specification uses the same or similar properties as defined by Schipér in [Sch06] for TO-multicast and group membership. In addition, we introduce properties specific to primary election based on leases.

4.2.1 Total Order Multicast

The main responsibility of the total order multicast component is to ensure that all processes in a process group executes events or messages in the same order. It exports two primitives:

in *to-multicast*(m) Multicasts a message m according to a total order.

out *to-deliver*(m) Executed for a message m according to the total order when the order was decided on.

The primitives guarantees the following properties:

TO1 Validity If a *group correct* process executes *to-multicast(m)* then it will eventually execute *to-deliver(m)*.

TO2 Uniform Integrity For any message m, 1) every process executes *to-deliver(m)* at most once and 2) only if m was previously *to-multicast* by some process.

TO3 Uniform Agreement If a process executes *to-deliver(m)* in a view v, then all *view correct* processes in v eventually executes *to-deliver(m)*.

TO4 Uniform Same View Delivery If two processes p and q executes *to-deliver(m)* while in view v for p and view v' for q, then $v = v'$.

TO5 Uniform Total Order If some process executes *to-deliver(m)* in a view v before it executes *to-deliver(m')*, then every process p in v executes *to-deliver(m)* before they execute *to-deliver(m')*

Property **TO4** ensures that a message that a *to-multicast* message is delivered in the same view for all processes. Furthermore, **TO3** states that as soon as one process delivers a message, then all correct processes in that

view will also deliver the message. Therefore, it is valid to *to-multicast* a message, change the view, and then *to-deliver* the message in the new view. This differs from view synchrony, part of the ISIS group communication toolkit [Bir93], where the multicast and deliver must occur in the same view. The validity-property, **TO1**, states that if a process which is faulty in a view executes *to-multicast(m)* it may or may not deliver m.

The definition of total order in **TO5** is gap free [DSU04] which is necessary to implement a replicated state machine on top of the TO-multicast abstraction. It is also necessary for the group join and leave abstractions presented in the next section. Without gap free delivery it is possible to reach a "contaminated" state where a faulty process in the current view TO-multicasts a message based on an incomplete history (i.e. it has delivered m_1 and m_3 but not m_2 which was delivered by the some other process). Also, note that a process can *to-deliver* m in v and m' in a new view v'. Thus, if q is not in v it can still deliver m' in v' and if p is not in v' it does not deliver m' even if it delivered m.

Example Traces

To get a more intuitive understanding of the specifications for TO-multicast, we introduce three example traces. Figures 4.2 and 4.3 has a valid execution (a) and two invalid executions (b) and (c) of the TO-multicast specification. In (a), $p1$ and $p2$ *to-multicast* messages $Op(x), Op(y)$ and $Op(z)$. The messages are *to-delivered* in the same order at all processes. Note that even though $Op(z)$ is *to-multicast* before $Op(y)$, the order agreed upon is x, y, z. This is correct since the total order property **TO5** only defines the delivery order and this order does not need to be in the same order as the multicast invocations (e.g. FIFO).

Trace (b) and (c) both violate the specification. Execution (b) is invalid since process $p3$ delivers $Op(x)$ out of order. That is, both $p1$ and $p2$ delivers $Op(x)$ first and then $Op(y)$, while $p3$ delivers $Op(y)$ before $Op(x)$ which violates **TO5**. In (c), the violation is more subtle, both $p1$ and $p2$ delivers $Op(y)$ before the view change while $p3$ delivers it after. This violates **TO4** since if one process delivers the message in a view, then all processes must deliver it in the same view.

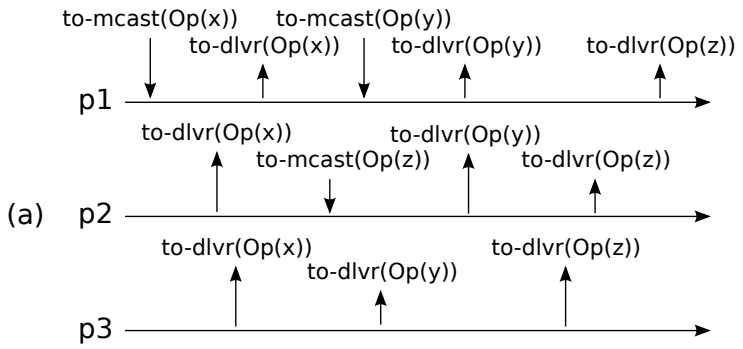


Figure 4.2: Trace of a valid (a) execution of *to-multicast* and *to-deliver* in a process group with three members.

4.2.2 Group Membership

The group membership module is used to initialize and disband process groups as well as reconfiguring their current view. Reconfiguration of a group is done by either joining or removing a process and increase the view's id. View ids must follow a total order to avoid that processes are added to an old view. The group membership specification focuses on mechanisms for adding and removing processes, not why they should be added or removed. For example, if a process has failed, an external mechanism such as a monitor service with a failure detector on all processes in the group, must find out if the process has failed and decide to remove it from the group.

The problem of changing the membership can be reduced to agreeing on a sequence of views. Since TO-multicast uses agreement to ensure a total order of messages, we can base the group membership specification on the TO-multicast specification. All operations affecting the state of the group must be performed atomically relative to any TO-multicast operations. If a TO-multicast operation and a group reconfiguration occurs at the same time (concurrently) it may lead to inconsistent state. To avoid this type of inconsistency we execute view change operations by using the TO-multicast module. Both specifications are therefore nearly identical.

The group membership module exports the following primitives:

in *init-group*(*members*) Initializes a new group with the given processes as the first view.

in *disband-group*() Disbands the group, this is the final operation in the group.

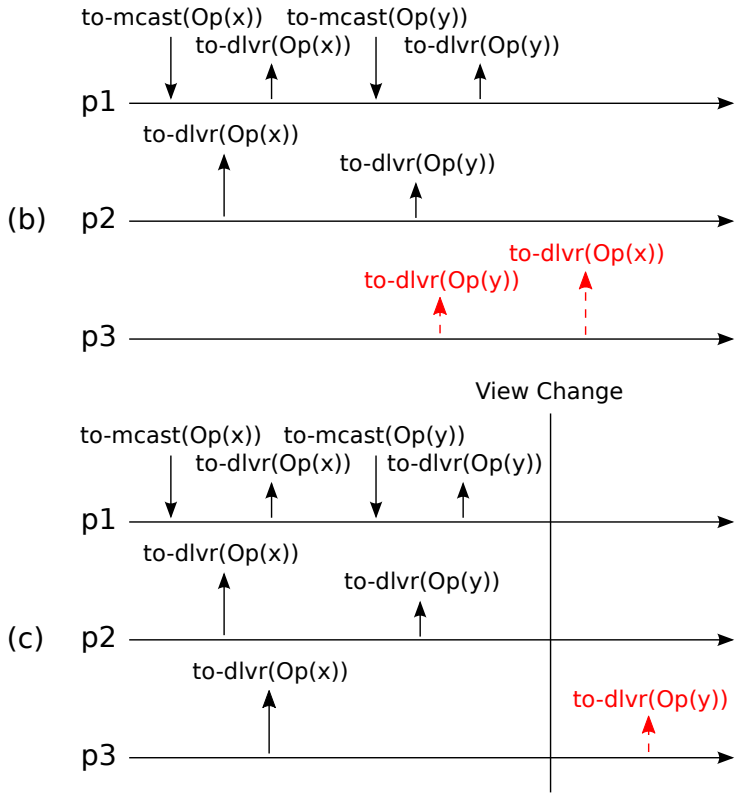


Figure 4.3: Trace of two invalid executions, (b) and (c), of *to-multicast* and *to-deliver* in a process group with three members.

in *join*(*process*) Adds a process to the group.

in *leave*(*process*) Removes a process from the group.

Group membership is defined using properties from [Sch06]. Note that our specification makes the reliance on $to-multicast$ and $to-deliver$ explicit and we have added properties for disbanding and initializing the group. Let GroupJoin(x) and GroupLeave(x) represent the messages sent when joining and leaving a group, respectively. Except for properties **G1** and **G2**, the group membership and TO-multicast specifications are almost identical. Further, note that property **G4** prevents a process from joining or leaving the group more than once. Thus, a process is not allowed to re-join.

G1 Group Initialize The initial view of a group is $v_0 = (0, S_0), S_0 \in \Pi$.
For every process p, its initial view v, is defined by the execution of

init-group or *join*(p), when $v \neq v_0$. For any p it is in v and either $v = v_0$ or there exists a process $q \neq p$ that installs v.

G2 Group Disband If the final view of a group is v_{end}, then for each process p in v_{end}, there is no view $v'_{end} > v_{end}$ and there is no process p executing *to-deliver*(m), for any message m, after it has executed *to-deliver*(GROUPDISBAND(q)).

G3 Group Validity If a group correct process p executes *to-multicast*(GROUPJOIN(q)) (or GROUPLEAVE(q)), then p eventually executes *to-deliver*(GROUPJOIN(q)) (or GROUPLEAVE(q)).

G4 Group Uniform Integrity For any process p, every process q executes *to-deliver*(GROUPJOIN(p)) (or GROUPLEAVE(p)) at most once, and only if *to-multicast*(GROUPJOIN(p)) (or GROUPLEAVE(p)) was previously executed.

G5 Group Uniform Agreement If a process p executes *to-deliver*(GROUPJOIN(q)) (or GROUPLEAVE(q)) in a view v, then all view correct processes in v eventually executes *to-deliver*(GROUPJOIN(q)) (or GROUPLEAVE(q)).

G6 Group Uniform Same View Delivery If two processes p and q execute *to-deliver*(GROUPJOIN(r)) (or GROUPLEAVE(r)) in view v for p and view v' for q, then $v = v'$.

G7 Group Uniform Total Order Let $op(p)$ denote either *to-deliver*(GROUPJOIN(p)) or GROUPLEAVE(p). If some process executes $op(p)$ in view v before it executes $op(q)$, then every process in v executes $op(p)$ before they execute $op(q)$.

Example Traces

Figure 4.4 shows a valid (a) and an invalid (b) execution of the group membership specification. Initially the group consist of $p1$ and $p2$, before $p1$ executes a join-message for $p3$. $p3$ becomes part of the group when it executes the join-message referencing itself. After the view has changed at $p1$, it triggers a leave-message for $p2$. In execution (b) $p1$ decides to stop the group by sending a *Disband*$()$ message. All processes delivers the message, but $p2$ delivers another message $Op(y)$ after delivering the *Disband*$()$. This violates property *G2*, since no messages are allowed to be delivered after a group has been disbanded.

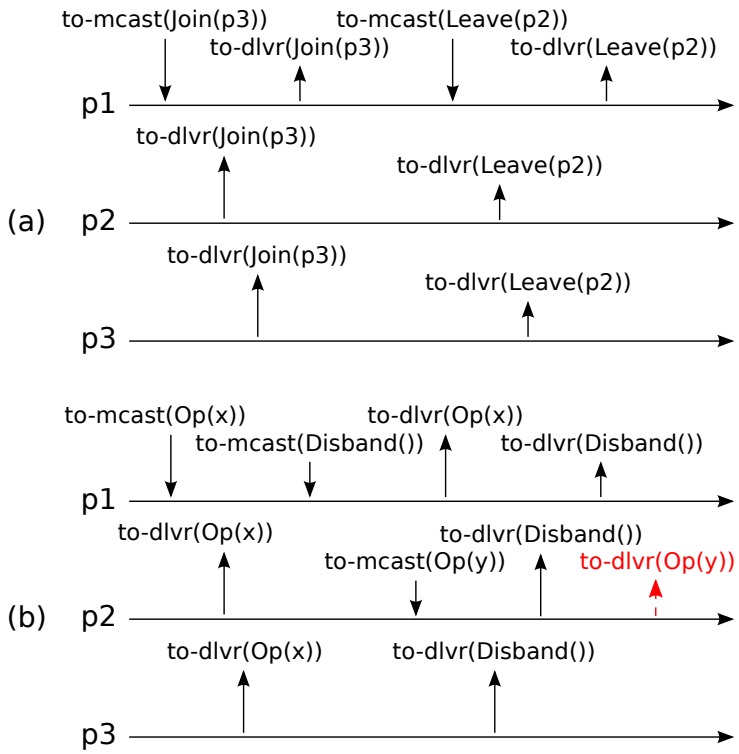


Figure 4.4: Traces of a valid (a) and invalid (b) execution of the group membership specification.

4.3 Implementation

In this section we introduce a solution to the total order multicast and group membership problems. We use primary/backup with a fixed sequencer [DSU04] to solve the total order problem. Only a single process at a time decides which order the multicasts should be executed in. Intuitively, this is simpler than destination agreement (Paxos) where all processes must agree on the order. However, primary/backup needs additional algorithms for handling primary failure and recovery.

Our implementation uses a lease to indicate the primary-role and handle primary fail-over. A lease is a time-based lock that guarantees mutual exclusion. This has two advantages over traditional leader election algorithms. First, due to the property of leases, at most one process may be a valid primary guaranteeing the exclusivity of the primary. Second, a failed primary will be replaced when the time limit of the lease expires through re-negotiation of the lease.

4.3.1 A TO-Multicast Algorithm

The total order multicast algorithm is executed within a process group. The processes in the group are called backups, where one backup is having the role of the primary. The primary is responsible for deciding on the message order and to make sure this order is maintained even when failures occur. The backups are storing the message order and are ready to take over the primary role when the current primary fails. The protocol is majority-based and can make progress with f failed processes in a process group with $2f + 1$ members. The algorithms below are divided into primary and backups and includes the recovery process.

Algorithm Outline. A single round of the TO-multicast algorithm has two phases (alg. 2 line 4-8). In the first phase, the primary receives a new TO-multicast request and broadcasts it to all backups in the current view. A backup that receives a TO-request adds it locally to a list of received requests and replies to the primary. The primary waits for a majority (or large enough quorum) of the group members to reply (alg. 3 line 15-18) and when a majority has acknowledged the request to the primary, the request is marked as *stable*. A stable request can be recovered if a minority of the processes fail, i.e. f failures from $2f + 1$ processes.

In the second phase, when the message is stable, the primary sends a notification to the backups that the message can be delivered to the application. Each message is associated with an *id* and a message can only be delivered at the backups when all messages with smaller id's have been delivered. Multiple rounds are chained together at the primary by simply increasing the message id (alg. 2 line 11-13).

Total Order. We define the message id as an element from $\mathbb{N} \times \mathbb{N}$. The first part of the id is called the *epoch* and the second part is the *version*. The *epoch* uniquely identifies the primary that multicasted the message and the *version* is monotonically increasing for each multicast. Primary epochs are necessary to handle recovery when the primary fails in the first phase of the algorithm and cannot complete the execution. The sequence of epochs may contain gaps, but must always increase. The sequence of version, on the other hand, must not contain any gaps, e.g. $1, 3$ is an invalid sequence. We define the total order $\prec_{id}$ on id's as follows: $id \prec_{id} id' \Leftrightarrow id_{epoch} \leq id'_{epoch} \wedge id_{version} < id'_{version}$.

Algorithm 2: Total order multicast algorithm at the primary.

1 $id \leftarrow (1,1)$ ▷ The id is a tuple with (epoch, version) totally ordered by $\prec_{id}$.
2 $delivered \leftarrow \emptyset$ ▷ Delivered messages.
3 $members$ ▷ Processes that are part of the current view.
4 **procedure** *toMulticast(id′,msg)* **do**
5 **if** $|members| = 0$ **then return**
6 **send** TOMulticast*(id′,msg)* **to** *members*
7 **wait until** TOMulticastAck*(id′)* **from** $\lceil \frac{|members|+1}{2} \rceil$
▷ Received ack from a majority, confirm that message can be delivered.
8 **send** TOMulticastDeliver*(id′)* **to** *members*
▷ Delivers the message internally (membership management) or to the application.
9 *deliver*(*msg*)
10 $delivered \leftarrow delivered \cup (id', msg)$
11 **procedure** *toMulticast(msg)* **do**
12 $id_{version} \leftarrow id_{version} + 1$
13 **return** *toMulticast*(*id*,*msg*)
14 **on receive** TOCatchUp*(id′)* **do**
▷ All entries which are newer than id'.
15 $entries \leftarrow filter(delivered, id' \prec_{id} id)$
16 **reply** TOCatchUpAck*(entries)*

Backups. A backup is responsible for taking over the primary role in case of a failure. It can additionally be used as a read-only source for in order message notification by application clients. The backup is storing and delivering multicast messages in the order decided by the primary. A message goes through three different stages: pending, confirmed and delivered. When the backup receives a TOMulticast()-message (alg. 3 line 18-21), it stores the multicasted message as pending. At this point, the message is not stable and a pending message can later be removed if it violates the delivery order in case of a new primary. A pending message becomes confirmed when the backup receives a TOMulticastDeliver()-message (line 22-26). A confirmed message can be delivered if all messages with a lower id has been delivered according to the total order $\prec_{id}$ (*tryDeliver* line 8-17).

Recovery and Catch-Up. When a primary fails a new primary must take over. Similarly, when a new process joins the group or one falls behind due to message loss or delays, it must catch-up with the messages delivered so far. For a backup to catch-up, it queries the primary about all messages multicasted after the backup's current id (alg. 4 line 13-17). The result is added to the current set of confirmed messages and then delivered if possible.

When a new primary gets elected (see 4.3.2), it executes the *becomePrimary*-method in alg. 4 with a new (larger) epoch. The recovery guarantees that the new primary has the latest state when it starts to TO-multicast new messages and that an incomplete multicast from the old primary is repaired. Unlike the catch-up of a backup, the new primary must get its state from a majority of backups. Given the quorum intersection property, reading from a majority will return all stable (majority is pending) and confirmed messages. If only a minority was reached by the incomplete multicast, the new primary may or may not see this message as pending. All pending messages are repaired by redoing the multicast. If a backup stores a pending message where the *id* violating $\prec_{id}$, it can safely discard it (after line 20).

Optimizations

The TO-multicast algorithm presented in the previous section can only execute a single operation at a time and relies entirely on the primary for

Algorithm 3: Total order multicast algorithm at the backup.

```
1  id ← (1,1) ▷ The id is a tuple with (epoch, version) totally ordered by ≺_id.
2  pending ← {}     ▷ Map from id → message for unconfirmed messages.
3  confirmed ← {}          ▷ Map from version → (id,message) for confirmed
   messages, sorted by version.
4  delivered ← ∅                          ▷ Delivered messages.
5  members              ▷ Processes that are part of the current view.
6  primary                              ▷ The current primary.
7  self                                 ▷ The current process.
8  procedure tryDeliver() do
9      version ← id_version
       ▷ If the next to be delivered has not been confirmed, try to catch-up.
10     if confirmed[version] = ∅ then catchUp()
11     while (id′, msg) ← confirmed[version] do
12         delete confirmed[version]
13         version ← version + 1
           ▷ Internal delivery excludes delivery to the application.
14         if not deliverInternal(msg) and self ∈ members then
15             deliver(msg)
16         delivered ← delivered ∪ (id′, msg)
17         id ← id′
18 on receive TOMulticast(id′,msg) do
       ▷ Only add to pending if we didnt deliver or confirmed this message.
19     if confirmed[id′_version] = ∅ and id ≺_id id′ then
20         pending[id′] ← msg
           ▷ Discard any message in pending where ≺_id is violated.
21     reply TOMulticastAck(id′)
22 on receive TOMulticastDeliver(id′) do
23     if id′ ∈ pending then
           ▷ Store the message from pending and id as confirmed.
24         confirmed[id′_version] ← (id′, pending[id′])
25         delete pending[id′]
26         tryDeliver()
```

Algorithm 4: Algorithms for catching up and recovery when becoming primary.

```
1  procedure becomePrimary(nextEpoch) do
       ▷ Retrieve the latest state from a quorum.
2      send TORecover(id) to members
3      wait until TORecoverAck(pending′,confirmed′) from ⌈(members+1)/2⌉
       ▷ For all unique messages, if at least one has confirmed, we can
       confirm.
4      foreach (id′, msg) ← confirmed′ do
5          confirmed[id′_version] ← msg
       ▷ Retry all messages that are pending but not confirmed.
6      retries ← ∅
7      foreach (id′, msg) ← pending′ if confirmed[id′_version] = ∅ do
8          retries ← retries ∪ (id′, msg)
9      tryDeliver()
       ▷ Block until all pending messages have been to-multicast.
10     foreach (id′, msg) ← retries do
11         toMulticast(id′,msg)
12     id_epoch ← nextEpoch
13 procedure catchUp() do
14     send TOCatchUp(id) to primary
15     wait until TOCatchUpAck(delivered′) from
16     confirmed ← confirmed ∪ delivered′
17     tryDeliver()
18 on receive Recover(id′) do
19     confirmed′ ← confirmed ∪ delivered if id_epoch ≥ id′_epoch and
       id_version ≥ id′_version
20     pending′ ← pending if id_epoch ≥ id′_epoch and id_version ≥ id′_version
21     reply RecoverAck(pending′,confirmed′)
```

backup catch-up. In this section we discuss a number of improvements to the protocol introduced above.

Pipelining. In the above protocol, TO-instances are executed sequentially. This simplifies the presentation and implementation, but increases the latency for each request linearly based on the queued messages. An alternative approach is to use pipelining, where a set of α instances are executed in parallel [LAB+06]. Each message is assigned a unique id among α instances by the primary. When all messages are stable at the backups, the primary can issue the next set of α instances. This scheme increases throughput due to the increased parallelism as well as reduces the average latency when requests are queued.

Batch operations. Each TO-multicast instance delivers a single message in order. By bundling several messages in a single instance the throughput can be significantly increased [MPP11]. The bundled messages are deterministically delivered in the same order at all backups based on the id assigned by the primary. The main effect from batching is a reduction in small messages being sent and received. Each message typically result in one context switch at the sender and one at the receiver. This issue is address specifically in Ring Paxos [MPSP10].

Conflicting operations. Two operations are *conflicting* when they depend on each other in such a way that concurrent access violates the consistency condition of the data. For example in a key/value-store with strong consistency (linearizability), two operations for the same key are conflicting. However, operations for two distinctly different keys are not conflicting and can therefore be executed concurrently. The idea of executing conflicting messages concurrently have been addressed through generic broadcast [Ped99].

Backup Catch-up. When a backup tries to catch-up to the latest delivered TO-multicast, it reads the state of the primary. This approach uses additional resources at the primary which may already be under load. An alternative is for the backup to collect the latest epoch and version from the primary and then select one or more backups to read the data from. By doing the catch-up in parallel, the period of time a backup is behind can be reduced.

Application State. When a process catches up or joins the group, it must replay the entire log of delivered messages. In a log with m messages, this grows as $O(m)$ which is highly impractical in a long-lived group. A solution is to take snapshots of the state, which acts as a summary of the log. A new process then only needs to retrieve the snapshot and recover all messages since the last delivered message in this snapshot. The cost is then reduced to the snapshot size, s, and the messages delivered since the snapshot was taken m_s, i.e. $O(s + m_s)$.

Piggybacking. In one TO-multicast round the primary broadcasts two messages, the initial TOMULTICAST() and a TOMULTICASTDELIVER() when the multicast is stable. The deliver message is only a notification and does not need to be sent directly. It can be implicit or piggybacked with the next TOMULTICAST(). When pipelining is used, it can be sent once, effectively delivering all messages for that round, at the end or beginning of each pipelining round.

4.3.2 Primary Election

The TO-multicast protocol described in the previous section lacks a mechanism to elect a new primary or to handle one that fails. This mechanism must guarantee that no two processes are primary at the same time. In addition, it must be possible to revoke the current primary if it fails. One way of ensuring this is to use leases [Lam96, GC89]. A lease is a time-limited lock which ensures a single process exclusive access to some resource. In our case, this resource is the primary role. If a process acquires the lease it becomes the primary for at least the time period of the lease. To extend the period, the process can renew the lease. If a lease is not renewed in time, e.g. due to the lease holder failing, another process can acquire the lease. In this section we introduce the lease negotiation protocol published in [KHSH], and make the necessary extensions to use it for primary election.

Lease Negotiation

The problem of acquiring a lease among a set of processes can be reduced to agreeing on which process should be the lease holder. Agreement can be implemented with Paxos, but unlike leases, Paxos requires the set of

processes to remember the result of a consensus instance indefinitely. A lease, however, is only valid for a limited time, t_{max}.

In Flease [KHSH], we introduce a lease negotiation algorithm which avoids using stable storage to store the result of each Paxos instance. The protocol is using the round-based register from [BDFG03] (see Sec. 2.3.2), which encapsulates a single Paxos instance, to agree on the lease owner in a set of processes. In addition, we take advantage of loosely synchronized clocks to decide if the lease is still valid.

The Round-Based Register. We introduced the round-based register in section 2.3.2 when describing Paxos. It abstracts a single paxos agreement instance using two operations: *read*(k) and *write*(k,*value*), where k is a round number proposed by each process. When a process executes read, it only succeeds if k is larger than some k' from a concurrent or previous invocation of *read*(k'). The result of the read is the stored value or $\perp$. Similarly, a write only succeeds if $k \geq k'$. Intuitively, a read followed by a write only commits if there were no other concurrent operations invoked with a higher round number. Specifically, the properties for a round-based register are [BDFG03]:

Reg1 Read-Abort If $read(k)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' \geq k$.

Reg2 Write-Abort If $write(k, *)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' > k$.

Reg3 Read-Write-Commit If $read(k)$ or $write(k, *)$ commits, then no subsequent $read(k')$ can commit with $k' \leq k$ and no subsequent $write(k'', *)$ can commit with $k'' < k$.

Reg4 Read-Commit If $read(k)$ commits with v and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' < k$.

Reg5 Write-Commit If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.

Algorithm 5 is an implementation of the round-based register. Each operation (read or write), are executed on a majority of processes and waits for either *ack* or *nack* (line 5-12 and 13-19). A single *nack* means

Algorithm 5: Round based register for process p_i, from [BDFG03]

```
1  read_k ← 0  ▷ The proposed k of the last successful read.
2  write_k ← 0  ▷ The proposed k of the last successful write.
3  value ← ⊥  ▷ The current value of the register.
4  members  ▷ The processes sharing the register.
5  procedure read(k) do
6      send READ(k) to members
7      wait until READACK(k,*,*) or READNACK(k) from ⌈(members+1)/2⌉
8      if received at least one READNACK(k) then
9          return (abort,⊥)
10     else
11         select the [ReadAck,k,k',value] with the highest k'
12         return (commit,value)
13 procedure write(k,v) do
14     send WRITE(k) to members
15     wait until WRITEACK(k,v) or WRITENACK(k) from ⌈(members+1)/2⌉
16     if received at least one WRITENACK(k) then
17         return abort
18     else
19         return commit
20 on receive READ(k) do
21     if write_k ≥ k or read_k ≥ k then
22         reply READNACK(k)
23     else
24         read_k ← k
25         reply READACK(k,write_k,value)
26 on receive WRITE(k,v) do
27     if write_k > k or read_k > k then
28         reply WRITENACK(k)
29     else
30         write_k ← k
31         value ← v
32         reply WRITEACK(k)
```

that another process tried to access the register concurrently with a larger round, k, and results in an *abort*. If no *nack* is received, a *read* returns a tuple (*commit*, v), where v is the current value of the register (either $\perp$ or the last value written). A write returns *commit*. We refer to [BDFG03] for an extensive description and a correctness proof of this algorithm.

The Flease Protocol. Flease, or a lease management service, exports a single operation: *acquireLease*(k), where k is the proposed round. An execution of *acquireLease*(k) either returns the lease $\lambda = (p, t)$ or $\perp$ when the lease could not be retrieved. p is the current process holding the lease λ, and t is the time when the lease expires. That is, the lease is *valid* while $\forall_{q \in members} c_q(t_{now}) \leq t$, where c_x is the local clock of a process x, t_{now} is the current time and *members* is the set of processes. The argument k, represents the same round number as passed to the round-based register and is used to ensure no other process tried to acquire the lease concurrently.

There are two system-wide constants, ϵ, which denotes the maximum clock skew between any two processes and t_{max}, the maximum time span of a lease. For the system to make progress, we assume that $t_{max} > \epsilon$. For practical reasons, t_{max} should be larger than the maximum round-trip time (RTT) to avoid that the lease times out before *acquireLease*(k) returns. Due to the loosely synchronized clocks, there is a time interval when a process does not know if the current lease holder still thinks that the lease is valid. During this *uncertainty interval* a new lease cannot be issued. From a process point of view, a lease with time $\lambda.t$ may still be valid when in the interval $\lambda.t < t_{now}$ and $\lambda.t + \epsilon > t_{now}$.

A lease protocol must guarantee that at most one process holds the lease. With an extension to consider views, the following safety property is ensured:

L1 Lease Invariant If a process p in v decides $\lambda = (p, t)$ then any other process in v will decide $\lambda' = (p', t')$, where $p = p'$ and $t' \leq t$ while $t_{now} < t$.

Algorithm 6 presents the flease algorithm with renewal and loosely synchronized clocks. The algorithm uses the round-based register to store the lease and to handle concurrent access. Similar to a single Paxos instance, the flease algorithm has a read-phase followed by a write-phase. A read followed by a write is necessary to ensure that 1) no process tried to acquire a lease concurrently and 2) to enforce a previous write that was

Algorithm 6: Flease with renewal and loosely-synchronized clocks

```
1  members ▷ The processes sharing the lease.
2  self ▷ A reference to the local process.
3  reg ← new Register(members) ▷ Initializes the round-based register.
4  procedure acquireLease(k) do
5      if reg.read(k) = (commit, λ) then
           ▷ Is it unknown if the lease timed out on all processes?
6          if λ.t < t_now and λ.t + ε > t_now then
7              wait for ε
8              return acquireLease(k')                    ▷ with k' > k
           ▷ Check if the lease is valid or if it should be renewed.
9          if λ = ⊥ or λ.t < t_now then
10             λ ← (self, t_now + t_max)
11         else if λ.p = self then
12             λ ← (self, t_now + t_max)
13         if reg.write(k,λ) = commit then
14             return (commit,λ)
15     return (abort,⊥)
```

incomplete (note the similarity with recovery of the primary in the TO-multicast algorithm). If the read from the register commits with a lease λ (line 6), the process either tries to overwrite the register with the already valid lease or it changes the lease.

With renewal and loosely synchronized clocks there are three cases that can execute when the lease changes. 1) The lease time, $\lambda.t$, is in the uncertainty interval where we cannot be sure if the current lease holder is still considering itself the owner, i.e. $\lambda.t < t_{now} \wedge \lambda.t + \epsilon > t_{now}$ (line 6-8). Then we wait for ϵ, to make sure that the lease has expired at all processes, and try to acquire the lease with a new $k' > k$. 2) The lease was never acquired (read returns $\perp$) or it has timed out, in which case the current process assigns itself as the new owner and $t_{now} + t_{max}$ as the time when the lease expires (line 9-10). 3) When the lease is still valid and the owner is the current process, the lease is updated with a new expiration time (line 11-12).

Lemma 4.3.1 *Algorithm 6 correctly implements property **L1 Lease Invariant**.*

Proof Proof by contradiction: Assume two processes p_i and p_j decide two different values $\lambda = (p, t)$ and $\lambda' = (q, t')$ with $\lambda \neq \lambda'$, $p \neq q$,

$t > t_{now}$ and $t' > t_{now}$, i.e. two different leases with different owners are both valid. Without loss of generality, we assume that $k' > k$ and that p decides λ after committing *acquireLease*(k). Also, q decides λ' after committing *acquireLease*(k'). Following Algorithm 6, q must commit *read*(k') before calling *write*(k', λ'). The read-abort property of the register (property **Reg1**) ensures that the *read* will commit because $k' > k$. Due to the write-commit property of the register (property **Reg5**), the *read* will commit with λ as this value was previously written by p. Depending on the value of $\lambda.t$, process q will make one of four decisions:

Case 1: $\lambda.t < t_{now}$ and $\lambda.t + \epsilon > t_{now}$ (λ is in the uncertainty interval, line 6-8)
The process waits for ϵ to make sure no process holding the lease still thinks its the owner, then we re-execute *acquireLease*(k''), where $k'' > k'$. Any of the cases below applies since $k'' > k'$.

Case 2: $\lambda.t \geq t_{now}$ (lease λ is still valid)

Case 2a: $\lambda.p \neq q$ (q does not hold the lease)
According to the algorithm, q will *write*(k',λ) and decide $\lambda' = \lambda$. However, this is a contradiction to the assumption that $\lambda' \neq \lambda$.

Case 2b: $\lambda.p = q$ (q holds the lease)
According to the algorithm, q will *write*(k',λ') and decide λ' with $p'_l = p_l$ and $t' > t$. However, this is a contradiction to the assumption that $p' \neq p$.

Case 3: $\lambda.t < t_{now}$ (the lease has expired)
In this case, q would *write*(k',λ') and decide λ' but is allowed to do so as we require q to decide λ only until $t_{now} > \lambda.t$. This is a contradiction to the assumption that $t > t_{now}$ and $t' > t_{now}$.

□

Protocol for Primary Election

Primary election tries to ensure that a single process in a process group is granted exclusive access to the primary role. The primary role is responsible for ordering requests as described in sec. 4.3.1. The primary election component uses the Flease protocol which guarantees exclusive access for a limited time period (lemma 4.3.1).

Algorithm 7: Primary election using flease.

```
1  self  ▷ The local process.
2  lease ← ⊥  ▷ The current lease value.
3  leaseid ← ⊥  ▷ The id of the current lease instance.
4  leases ← {}  ▷ A map leaseid → lease.
5  procedure getPrimary(view) do
6      if view_id ∉ leases and view_id > leaseid then
7          leases[view_id] ← new Lease(view_members)
8          leaseid ← view_id
9      k ← (t_now, self)
10     lease ← leases[leaseid].acquireLease(k)
11     if lease = (commit, λ) and λ.p = self then
12         becomePrimary(k)
```

From algorithm 7, a process executes *getPrimary*(v) to try to take ownership of the primary lease in a view v. For each installed view which the process is part of, a new lease instance is created (alg. 7 line 6-8). The instance is uniquely identified by the view id. Any incoming lease requests for unknown lease instances are discarded (results in a timeout at the requester when reading or writing the round-based register).

Allowing two concurrent primaries is necessary to ensure progress when the view changes. This does not violate the lease invariant since each view is associated with a different lease instance. For example, if the current primary fails during a view change operation, a majority may still be in the old view. Any process in this majority will try to become primary in the old view while one or more processes are already trying to acquire the primary lease in the new view. If a process becomes primary in the old view, it will re-execute the view change operation as the last operation in the view. From the agreement property of total order multicast, **TO3**, we know that all processes will eventually execute the view change operation and become part of the new view. The primary in the old view will not be able to execute any new operations in the old view since it is invalid after the view change.

Total order of k. Please requires that each process trying to acquire a lease always increases k and that k is totally ordered. This also applies to new processes joining an existing group. In order for these processes

to use an increasing value of k we use the current time, t. Additionally, to avoid that two processes uses the same k, we use the id of the process (alg. 7 line 9). The total order of k is defined as follows: if $k = (t, pid)$ and $k' = (t', pid)$, then $k < k' \Leftrightarrow (t < t') \vee (t = t' \wedge pid < pid')$

4.3.3 Group Membership Algorithms

The algorithms for membership management implements the four primitives: *group-init*, *group-disband*, *join* and *leave* specified in Sec. 4.2.2. The specification separates the mechanism for reconfiguration (join and leave) and the detection of failures. Each join or leave must therefore be executed by an external user. To simplify the algorithms a reconfiguration from one view to another only involves a single process joining or leaving. Thus, a quorum always overlaps between the current view and the next view.

The sequence of group membership changes are agreed on by using the TO-multicast implementation in algorithm 4.3.1. The view is internal state of the group, and all group membership operations are intercepted by the delivery method (algorithm 4.3.1 line 9) to avoid that the application receives the message.

Initialize and Disband. A group is initialized in the first view with a set of processes $S \subset \Pi$ by some external process, p_e. p_e assigns the group a unique id which can later be used to identify the group. The initialization of the group (alg. 8 line 4-6, 9-12) assumes that a majority of the processes are available. A majority is sufficient for a process to acquire the primary lease in the initial view. The process executing *initGroup*, p_e, broadcasts a prepare message and waits for a majority of members to acknowledge. When all acks have arrived, p_e knows that the processes are able to decide on a primary. A process receiving the init assign the members to the first view and sets the group id before trying to become primary. After a process has acquired the primary lease it can start executing TO-multicast requests.

To disband a group, the members must agree to stop accepting requests. This is achieved by executing TO-multicast with a disband-message (alg. 8 line 7-8, 13-15). When a backup receives the disband message it can shutdown itself. The primary, however, must still be alive to resend messages until all backups are shutdown. To avoid delivering any more TO-multicasted messages, the members are reset.

Algorithm 8: An algorithm for initializing and destroying a group.

```
1  members ← {}  ▷ The processes currently in the group.
2  viewid ← 0  ▷ The current view id.
3  groupid ← ⊥  ▷ The identifier for the group.
4  procedure initGroup(members′,groupid′) do
5      send GroupInit() to |members|
6      wait until GroupInitAck() from ⌈(|members|+1)/2⌉
7  procedure disbandGroup() do
8      toMulticast(GroupDisband(p))
9  on receive GroupInit(members′,groupid′) do
       ▷ Try to become primary, start executing TOMulticast() requests.
10     members ← members′
11     groupid ← groupid′
12     getPrimary((viewid, members))
13     reply GroupInitAck()
14 on deliver GroupDisband() do
       ▷ No members prevents any new operations in this group.
15     if isBackup() then shutdown process else members ← ∅
16     return true
```

Join and Leave. The algorithms for join and leave (alg. 9) uses the TO-multicast to agree on the view change. A process that delivers the join-message adds the new process to the set of members and increases the view id. Similarly, when delivering the leave message, the leaving process is removed from the members and the view id is increased. In both cases, all processes tries to become primary in the new view. This will succeed when a majority of processes have delivered the join message (respt. leave message).

The joining process will initially not know that it is part of the new view until the first TO-multicast message arrives. When this occurs, it recovers any application state from the primary or the backups and then uses the catch-up mechanism to re-execute all messages delivered in the view.

Algorithm 9: The algorithm for adding and removing processes from a group.

```
1  members ▷ The processes currently in the group.
2  viewid ▷ The current view id.
3  procedure joinGroup(p) do
4      toMulticast(GroupJoin(p))
5  procedure leaveGroup(p) do
6      toMulticast(GroupLeave(p))
7  on deliver GroupJoin(p) do
8      if p ∉ members then
9          members ← members ∪ {p}
10         viewid ← viewid + 1
           ▷ Try to become primary in the new view.
11         getPrimary((viewid, members))
12     return true
13 on deliver GroupLeave(p) do
14     if p ∈ members then
15         members ← members \ {p}
16         viewid ← viewid + 1
17         getPrimary((viewid, members))
18     return true
```

4.3.4 Message Complexity Analysis

We analyze the cost of the primary/backup approach for the common case, not including recovery or catch-up. There are two metrics which are interesting: number of messages and the number of message delays. The presented TO-multicast protocol needs a single round-trip from the primary to the backups before a message is stable and can be acknowledged back to the client. This results in two message delays and $2(|S| - 1)$ messages, where S is the set of group members. The latency depends on the slowest backup and link in the majority of S. The lease-algorithm used for primary fail-over is based on Paxos. In the failure-free case, it requires two round-trips (4 message delays) to negotiate a lease with a message cost of $4(|S| - 1)$.

4.4 Related Work

The significant contributions to total order broadcast and group communication systems up until 2000 was summarized in two surveys [DSU04] and [CKV01]. The main work after these surveys has been on runtime reconfiguration [LAB+06, LMZ10], implementing and deploying Paxos [CGR07] and byzantine fault tolerance [Ser10].

A primary/backup-based total order broadcast protocol has been specified and evaluated in a production environment by Junqueira et. al. in [JR09, JRS10, JRS11]. This protocol has been used in the implementation of the ZooKeeper coordination service [HKJR10]. Our protocol differs in two main ways. First, we use a lease-based leader election protocol to guarantee a unique primary. Their protocol relies on a bullying-like algorithm where the process with the highest identifier is assigned the primary role. It is sufficient if a quorum including the process believes that this is the highest identifier to make progress. Second, they do not describe how to reconfigure the system at run-time.

The specification for total order multicast and the dynamic group membership was proposed by Schipér in [Sch06]. We have used the same properties for both safety and liveness but introduced a different solution. Whereas the implementation of the specification in [Sch06] is based on an arbitrary agreement abstraction, our implementation uses primary/backup with strong primary election. We present a complete solution including algorithms for TO-multicast, recovery of the primary

and catch-up of backups and finally group membership.

Petal's global state manager (GSM) [LT96] allows a single add or remove [LAB+06]. This ensures that the quorums of two consecutive configurations always overlap. For example, any quorum of $\{p, q, r\}$ is overlapping with any quorum of $\{p, q, r, s\}$ or $\{p, q\}$. Unlike our Primary/Backup-based solution, the GSM is using a variant of Paxos.

4.5 Summary and Discussion

We have presented a process group specification and implementation of total order multicast with support for run-time reconfiguration. The protocols are based on primary/backup with leases for primary fail-over. A design goal was to separate policies from mechanisms [Sch06]. For a process group, this is significant since we can leave out any components for automatically detecting failures and deciding when a process should be removed from the group. The responsibility of adding and removing members is instead on the application. Using probability thresholds or depending on timeouts to decide on failures is hard since processes could be temporarily unavailable due to network failure or just slow because of, for example, a scheduled OS operation. For these reasons, explicit removal of servers in data center environments is common in practice [LM10, FHIS11].

There are two main reasons why we developed a primary/backup-based solution: simplicity and performance. First, with a primary that orders the sequence of requests and decides when an operation is completed, a more complicated agreement algorithm such as Paxos is avoided [CGR07]. Second, with a strongly elected primary through, for example, leases [CGR07], the primary always knows the latest state of the system. Therefore, a client executing an operation for observing the current state, e.g. a read-request, can be served directly by a primary without asking the other servers. With destination agreement, a read-request always requires at least one round-trip when contacting the other group members [BDFG03].

However, primary/backup have problems with load balancing, since the primary must send out and wait for replies from all backups. This result in an uneven utilization of the machines and may also make the primary overloaded faster during high load. A solution to this problem is to have a rotating primary [DSU04]. Furthermore, unlike Paxos, we

must explicitly provide algorithms for backup recovery and primary fail-over. For this, we have used the Flease algorithm [KHSH] which ensures a process exclusive access for a limited time period. Setting the lease time-out must be done by care, since if the primary fails right after acquiring the lease no operations are possible until a new process can take over the primary-role.

Chapter 5

Routecast and Partition Management

The architecture presented in chapter 3 has four main components: process groups, routing service, partition management and routecast. In the previous chapter we provided a specification and implementation for a dynamic process group. A single process group reliably stores state for a partition or a subset of the name space. In this chapter, we will describe how to maintain partitions.

Partition maintenance entails operations for modifying the name space, i.e. change which elements goes into what partition, and changing the assignment from partition to process group. These operations makes it possible to scale up and down the system dynamically. In order to consistently access state stored by a partition, even during reconfiguration, we introduce the *routecast*-primitive. *routecast* guarantees that all messages for a single name space element is delivered according to a total order.

In this chapter we specify and provide algorithms for the partition management and the *routecast*-primitive. Before introducing the specification in section 5.2, we start with a presentation of the prerequisites and the system model in 5.1. This is followed by the algorithms solving the problem defined in the specification in section 5.3, a description of how to handle application state 5.4 in and a summary and discussion in section 5.5.

5.1 Preliminaries

Process Groups. A group consist of a subset of processes from Π. Groups are dynamic, i.e. processes can join and leave. Processes in a group can fail and recover. A specific group configuration is called a *view*, and defined by a group id, a view id and a set of member processes, $(groupid, viewid, S) \in \mathcal{V}$, where $\mathcal{V}$ is the set of all views. The set of process groups is denoted $g \in \mathcal{G}$, where g is the latest view of a group.

Name Space and Partitions. Let the name space, D, be the elements of a totally ordered set and let P be the set of all unique partitionings of D. Furthermore, let $P_D \subset P$ be the set of all valid partitionings of D and $\rho \in P_D$ be one such partitioning. A partition p has a *range*, a version and an owner, $p = ([a, b), version, owner) \in \rho$, such that $a, b \in D$ and $a < b$. An element x is in the range $[a, b)$, if $a \leq x < b$. We say that $[a, b)$ *covers* x or that x is covered by $[a, b)$. Let $\perp$ be the smallest element (inclusive) and $\top$ an element larger than any element (exclusive) in D. The owner refers to a group that stores the partition. A partition with version k is denoted p^k or $[a, b)^k$ and it's owner, $owner(p^k)$ or p^k_A, if a group A is the owner. Furthermore, we define an ordering $\prec_x$ for partitions that covers $x \in D$. That is, if two partitions p and q existed that both covered x, then $p^k \prec_x q^{k'}$ when $k < k'$.

Partition Assignment. For any valid partitioning $\rho \in P_D$, there exists an assignment relation from ρ to the set of views $\mathcal{A} : \rho \to \mathcal{V}$. A process group in a certain view is *responsible* for an element $x \in D$ if it stores the partition covering x in that view. Similarly, it is responsible for a partition $p \in \rho$ if it stores p. The other way around, we say that a partition is *assigned* to a view. We call the change of assignment of a partition from one process group to another a *handover*. That is, a handover is between views with different group IDs. Note that any change to the view also affects the assignment $\mathcal{A}$.

Pointers. A pointer is a reference to a partition and the processes storing the partition. Due to changes in the assignment of a partition to the process group, the process group or to the partitioning itself, pointers can become outdated. A *correct* pointer is referencing the latest version of a partition.

5.2 Specification

We introduce three operations for performing changes to the partitioned name space: *split*, *merge* and *handover*. The *routecast*-primitive, forwards a message towards the owner of a partition containing a given element from the name space. A *routecast*-execution ends with *rc-deliver* which delivers the message to the application.

5.2.1 System Initialization

We provide a single operation to initialize an instance of RECODE. This is executed at a single process group and defines the name space range.

in $initialize(\bot, \top)$ Initializes the name space with a single partition covering $[\bot, \top)$.

There is a single property for the initialization.

P0 Initialization If a process group A executes *initialize*, it contains the initial state consisting of a single partition $([\bot, \top), 1, A)$.

5.2.2 Partition Management

Given the definitions of the name space, we describe *split*, *merge* and *handover*. Informally, a split divides a single range into two new ranges and a merge combines two ranges into a single range. Both split and merge-operations applied to a partitioning ρ results in a new partitioning $\rho' \in P_D$. The *handover* changes the assignment of partitions to groups.

in $split(x, p)$ Split divides the range of a partition $p = ([a, b), k, A)$ into two partitions $[a, x)$ and $[x, b)$, where $a < x < b$ and the position of the split is the element $x \in D$. The result is a new partitioning of D and a change in the assignment since the divided range does not exist in the new partitioning. The two new partitions are assigned to the same process group as before the assignment change.

in $merge(p, q) \rightarrow r$ Merge combines two consecutive partition ranges $p = ([a, x), k, A)$ and $q = ([x, b), k', A)$ into a partition with a single range $[a, b)$, where $a < x < b$. The result of a merge is a new partitioning and assignment in $\mathcal{A}$. The same group is responsible for both p and q before the merge and the new partition after the merge.

in *handover*(x, k) Updates the process group responsible for a partition p covering a name space element x to a new group $A \in \mathcal{G}$. k is a proposed version which is assigned as the new partition version after a successful handover. Thus, if $owner(p) = B$ before the handover, then $owner(p) = A$ after the handover. This modification results in a new assignment in $\mathcal{A}$.

A valid partitioning $\rho \in P_D$ has the following properties:

P1 No gaps The union of all partitions $p \in \rho$ must cover the set D. That is, $\bigcup_{p \in \rho} = [\bot, \top)$

P2 No overlaps The intersection between any distinct pair $p, q \in \rho$ must be equal to the empty set. That is, $p, q \in \rho$, $p \cap q \neq \emptyset \leftrightarrow p = q$.

By guaranteeing that there are no gaps and that no ranges overlap, an element in D is always covered by exactly one range for any valid partitioning in P_D. For example, if $D = \{1, 2, 3\}$ then $P_D = \{123, 1/23, 12/3, 1/2/3\}$, where / indicates a partition into subsets. $13/2$ is not in the set since P_D only contains partitions with monotonically increasing and ordered subsets. Using range notation, the partitioning $1/23$ is expressed as $[1, 2), [2, \top)$.

The *split* and *merge* operations are executed at a single process group with local coordination. The operations are deterministic and succeed as long as the preconditions are correct. To avoid any coordination for a *merge*(p, q), the same process group must be responsible for both p and q. Similarily, for a $split$, the executing process group is responsible for both of the resulting partitions.

P3 Split and Merge Termination A process group that initiates a split or merge operation eventually terminates the operation at most once.

The *split* and *merge* operations do not violate the overlap and gap requirements of a partitioning. Assume that a split is performed in a partitioning $\rho \in P_D$, and results in a new partitioning ρ'. ρ' is valid since the partition that was split is removed and the intersection of the new partitions is empty. Similarly, merge removes the overlapping ranges and ρ' has no gaps since the start and end of the new range is aligned with the start and end of the union of the merged ranges.

In order to change the responsibility of the partitions from one process group to another, we introduce the *handover*-operation. This operation must be atomic to avoid violating the requirement of consistent data access. If this operation is not atomic, two different groups would be responsible for the same or an overlapping range at the same time. This could lead to conflicting operations on elements in the range. The main safety property of the service defines a correct assignment as follows:

P4 Exclusive Assignment No two process groups are responsible for the same element $x \in D$.

Essentially, property **P4** states that for any possible partitioning $\rho \in P_D$, when a partition in ρ is assigned to some process group in $\mathcal{G}$, then this is the only group it is assigned to. That is, two groups are never allowed to be responsible for the same partition, but the same group can be responsible for more than one partition. A partitioning service where $\mathcal{A}$ is not well-defined, i.e. allowing a single partition to be mapped to more than one process, is possible by modifying the strictness of this property. For example, the definition could be that eventually a single group is responsible for the partition. However, that will also violate the requirement of total order which can lead to, for example, create-create conflicts where the same element is created in two different groups.

We introduce two additional properties which guarantee that any handover that has started will eventually terminate, and that a handover is only decided on once.

P5 Handover Validity For any handover, it is initialized by some process and it terminates at most once.

P6 Handover Termination A handover of a partition p between two process groups $A, B \in \mathcal{G}$, eventually terminates with either A or B responsible for p.

Property **P6** guarantees that a successful assignment change for a partition $p \in \rho$ leads to a new partitioning ρ'. The change itself must be atomic to avoid violating property **P4**, but the new owner only needs to be responsible for p at the time of termination. **P5** avoids that a handover is decided on twice, effectively, re-assigning a partition to the same owner.

5.2.3 Routing Service

Routing is a process which uses pointers to redirect a message towards an owner. $(key, msg) \in \mathcal{M}_{\mathcal{R}}$ is a routed message in the set of routed messages, where $key \in D$ and msg is from $\mathcal{M}$, the set of valid messages. The routing service is used by the application or by RECODE directly to route a message towards a process group responsible for a key. To decide how to route, the service maintains a table from partition to responsible group. A process group must be able to update the service to reflect the current partitions owned by the group. The state of the routing service is updated in a best effort manor and does not need to be consistent with the partitioning state maintained by the process groups. In addition to routing, the service also maintains a table for looking up the groups currently in the system by using the group id. This is necessary to give an entry point for the application to find groups.

in *route*(k, m) Forwards a message m to the process group responsible for the partition that covers k.

out *route-receive*(*x*,*m*) Executed when m is received at a process in a process group responsible for the partition that covers x.

in *update-partitions*(*partition*) Used to update the partition state of the routing service. The *partition* has a range, version and an owner.

in *update-groups*(*group*) Updates the state on groups. The *group* is a tuple with $(groupid, viewid, members)$.

in *lookup-group*(*groupid*) Finds the view registered with the given group id.

out *lookup-group-ack*(*g*) Invoked as a response to *lookup-group*. Returns a tuple $(groupid, viewid, members)$.

Routing terminates with the execution of *route-receive* at some process in a process group. The execution is reliable as long as the initiating process is correct[1]. However, there is no guarantee that *route-receive* is executed only once for a message. At most once semantics for routed messages is difficult since the group responsible for x may change over time, unlike reliable messaging between two pre-defined processes. The routing service must ensure the following property:

[1]Reliability is achieved by re-trying the route-request until an ack is received.

R1 Eventual Routing Termination If a correct process executes *route*(*x*,*m*), then some process q in a process group $g \in \mathcal{G}$ eventually executes *route-receive* if $x \in p$ and $owner(p) = g$.

R1 implies that the routing service eventually knows which process group is responsible for a partition.

Routecast. The routing service enables key-based routing, given a key a message is forwarded towards a process responsible for a partition covering the key. In a system with replicas where several processes are responsible for the same partition, a request requires coordination to provide consistent access. We introduce the *routecast*-primitive which is based on the *route*-abstraction, but provides stronger guarantees. The module exports a *routecast* and the corresponding *rc-deliver*-primitives:

in $routecast(k, m)$ Forwards a message m towards the process group responsible for k.

out $rc\text{-}deliver(k, m, p)$ Delivers the message m at each member of a process group A responsible for the partition $p = ([a, b), x, A)$ where $k \in [a, b)$.

The *routecast*-primitive ensures that messages for a given key k is delivered in a total order at the process group currently responsible for the partition covering k. We call this property *partitioned total order delivery*, and it must be guaranteed independent of any partition management operations.

Partitioned Total Order Delivery. Routing finishes with the execution of *rc-deliver*. The main requirement of Recode is that this delivery is according to a total order. A single process can achieve a total order by delivering messages in the order they are received by the process (e.g. a FIFO queue). A process group can also deliver messages according to a total order by using TO-multicast. Thus, using *routecast* to deliver messages for an element in the name space for the initial configuration with a single partition is an extension of TO-multicast. However, we also want that messages delivered for elements in partitions created after a split, merge or handover operation are delivered in a total order. Below we specify the properties for *rc-deliver*.

PTO1 Partitioned Validity If a *correct* process executes *routecast*(x, m), then some process group A eventually executes *rc-deliver*(x,m,p^k) in a partition p^k where $owner(p^k) = A$.

PTO2 Partitioned Integrity A routed message (x,m) is only rc-delivered if some process executed *routecast*(x, m) and it is rc-delivered at most once.

PTO3 Same Partition Total Order For any pair of routed messages (x,m) and (y,m') that are rc-delivered in a partition p, there exists a total order $\prec_p$ such that if (x,m) $\prec_p$ (y,m'), then *rc-deliver*(x,m,p) is executed before *rc-deliver*(y,m',p).

PTO4 Last Partition Delivery If a process group invokes *rc-deliver*(x,m,p^k), then there exists no partition $q^{k'}$, where $x \in q^{k'}$ and $k' > k$.

PTO5 Partitioned Total Order For any pair of routed messages (x,m) and (x,m'), there exist a total order $\prec_x$ such that if (x,m) $\prec_x$ (x,m') then *rc-deliver*(x,m,p^k) is executed before *rc-deliver*(x,m',$q^{k'}$), where x is covered by p, q and $k \leq k'$.

These properties define that there is a total order on the partitions that covers x. That is, if p^k and q^{k+1} both cover x, than any routed message for x is delivered to the owner of the partition according to a total order. Furthermore, if two routed messages for x and y are delivered within the same partition, then they are also delivered in a total order. Because of **PTO3**, the delivery to elements in the same partition become dependent. A weaker specification could replace this property with one that makes delivery to different elements independent (parallel). A partition management operation is always in conflict with the delivery to an element in the partition.

5.2.4 Example Traces

An execution is a trace of operations executing at the process groups of the system. In this section we provide some examples of valid and invalid traces to illustrate the specification properties.

Figure 5.1 contains two valid partition management traces and one invalid. (a) starts the system and perform a number of splits on the initial partition $[\bot, \top)$. (b) builds on (a) and shows two handovers from $p1$ to

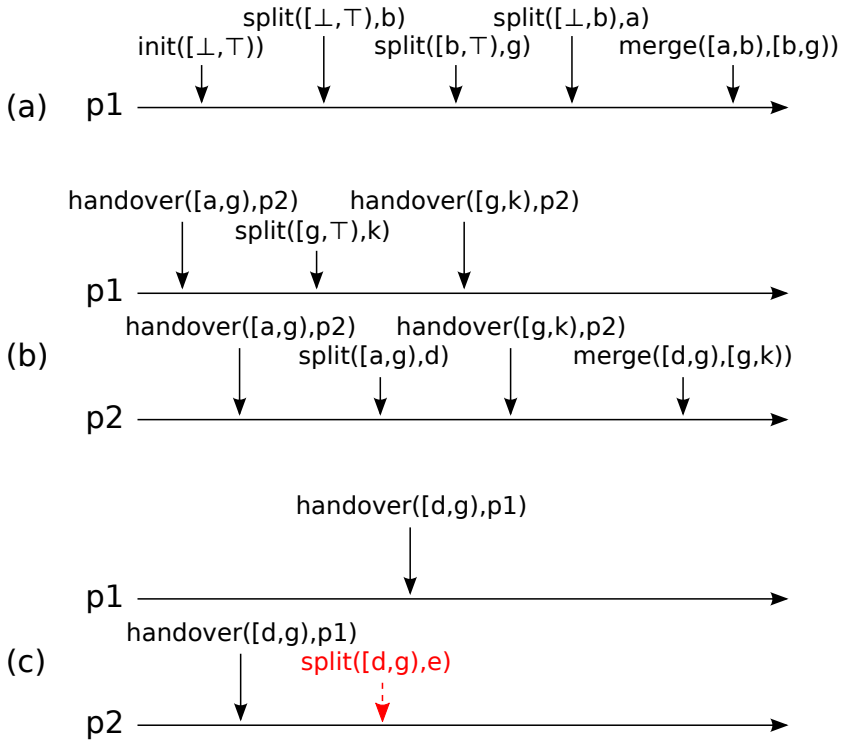


Figure 5.1: Two valid (a) and (b) and one invalid (c) trace of partition management.

$p2$ of $[a,g)$ and $[g,k)$. When $p2$ is responsible for both $[a,g)$ and $[g,k)$ it is allowed to merge these two partitions into $[a,k)$. When (b) has finished $p1$ is responsible for $[\bot,a)$ and $[k,\top)$ while $p2$ is responsible for $[a,d)$ and $[d,k)$. This is a good example of how the system can evolve and allocate partitions at new processes.

In the final part of the trace (c), $p2$ tries to split $[d,g)$ after it has handed over the same partition to $p1$. This violates property **P4 Exclusive Assignment** since both $p1$ and $p2$ thinks they are responsible for $[d,g)$.

Figure 5.2 illustrates the PTO-properties. We start these traces after (b) from fig. 5.1 with the following responsibilities $p1 : [\bot,a)^3, [k,\top)^4$ and $p2 : [a,d)^6, [d,k)^7$. The arrows for an *rc-deliver* are annotated with the result of executing g, i.e. $k.f(x,msg)$, where k is the current partition version. The versions of the partitions are derived from the partition management operations.

The first trace (a) is showing an execution satisfying the PTO-properties. However, note that $p2$ delivers in two different partitions which also

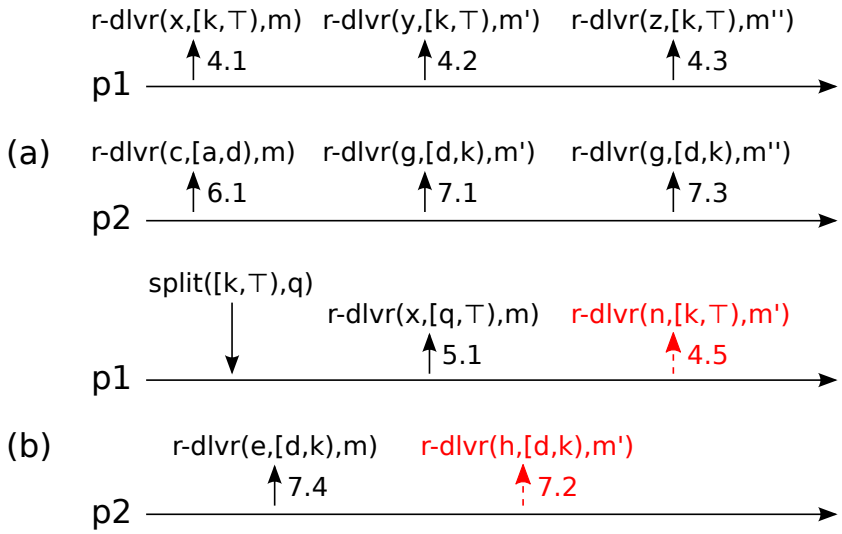


Figure 5.2: Traces illustrating the PTO-properties.

shows from the result of g. In addition, there is a gap of g's result in the deliveries of m' (7.1) and m'' (7.3). In trace (b) we can see the interaction between message delivery and partition management. This leads to a violation of **PTO4** at $p1$ which tries to *rc-deliver* in $[k, \top)^5$ after the creation of the overlapping partition $[k, q)^5$ through the split of $[k, \top)^5$. Furthermore, p violates property **PTO3** by delivering m' in 7.2 after the delivery of m in 7.4.

5.2.5 Discussion

A service that correctly implements the partition management properties can dynamically grow and shrink the number of groups responsible for partitions by 1) splitting and merging partitions and 2) handing over partitions between nodes. For the handover, property **P4 Exclusive Assignment** defines an exclusive leader for a partition. This is a crucial property for the specification of partitioned total order delivery. The specification restricts a service to only deliver messages for a given name space element according to a total order. This is necessary to provide strong application-level semantics when accessing a key even after a name space change.

5.3 Implementation

This section presents an implementation of the routecast-primitive and the partition management. The solution uses the process group from the previous chapter to handle process membership and TO-multicast. The distributed system model is partially synchronous and we assume quasi-reliable channels between processes. Process groups communicate by message passing, i.e. a member of a group sends a message directly to a member of the other group. The channel between two processes in different groups has the same properties as the channels between processes in the same group. A group's address consists of it's view and a globally unique id selected when first initializing the group.

If a group A needs to send messages to another group B, A must know at least one reachable process from B. We assume two strategies to handle changes to B. First, the sending group tries to keep references up-to-date by requesting the remote group's current state periodically. Second, if the receiving group changed too fast for the reference updating mechanism to fail, the routing service can be used as a fallback. We use the *update-groups*-operation to make sure that the routing service has an up-to-date view of each group's state.

5.3.1 Initialization

The initialization of the system creates a single instance covering the defined name space. Algorithm 10 uses the *to-multicast*-primitive to execute this operation reliably in the process group that was chosen to contain the first partition. The *partitions*-table uses the range as a key to look-up (*range*, *version*, *owner*)-tuples representing a partition. $self$ is a variable used to access the current group state: (*groupid*, *viewid*, *members*).

Correctness. The initialization method executes at a single process group and creates a partition range that covers the entire name space (line 3-7 10). Property **P0** is trivially satisfied from this initial state and the properties of TO-multicast. **P1,2** and **P4** are also satisfied since there exist no gaps, no overlapping partition ranges and only a single partition owner.

Algorithm 10: Initialization of a RECODE system instance.

```
1 partitions ← {}                          ▷ range ↦ partition
2 self          ▷ Current view of the group: (id, viewid, members)
3 procedure initialize(⊥,⊤) do
4 |   to-multicast(INITIALIZE(⊥,⊤))
5 on to-deliver INITIALIZE(⊥,⊤) do
6 |   partitions[[⊥,⊤)] ← (⊥,⊤,1,self)
7 |   update-routers((⊥,⊤,1,self))
```

5.3.2 Routing Service

The routing service maintains state of the current partition to process group assignment and the groups currently available in the system. How to implement a routing service that satisfies the property **R1 Eventual Routing Termination** has been explored in previous work [SMK+01, RS04, MD88]. Therefore, instead of presenting algorithms for an eventually consistent routing service, we discuss different routing topologies that are sufficient to satisfy **R1**. Since the state does not have to be consistent with the latest version maintained by the process groups, the partition state can be pushed to the service on any change. For example, see the call to *update-partitions* at line 5 in 10. To update the service on group changes, the algorithms presented in Chapter 4 are augmented to publish to the routing service when the group is created, destroyed and reconfigured.

For reliable use of $route(x, m)$ such that it eventually terminates, it needs to be executed more than once when the original message is lost. This leads to duplication, i.e. the owner of x receives and executes *route-receive* multiple times. There exist several approaches to reliable messaging [LJ06]. One way is to introduce and keep track of version numbers that are increased by the originator for each unique *route*-request. The receiver uses the version to avoid executing *route-receive* if the same message arrives twice. The other alternative is to handle duplication at the application level. For example, the handover algorithm (section 5.3.4) takes care of concurrency or duplication by increasing the version of the maintained state.

By creating a separate routing module, we achieve more flexibility. As long as the service guarantees eventual termination, any topology can be used for routing. The service is essentially caching the current state which is maintained reliably by the process groups. This separation is

similar to PNUTS [CRS+08] from Yahoo!. However, unlike our approach, their partition management is not completely decentralized. The routing function can be implemented as a library at the application clients, as a separate service with dedicated machines or as part of the processes in the process group and the topology can be of any type, e.g. ring (DHT), a tree (DNS) or a complete mapping table, as long as a routed message eventually arrives at the responsible group.

Ring. A ring-topology has forward and backward pointers between each process. In the worst case, routing a message takes $O(n)$-hops, where n is the number of processes in the system. For many applications the latency this incurs is unacceptable [DHJ+07]. Therefore, to reduce the number of hops additional pointers are added to achieve $O(\log n)$ or even $O(1)$ routing. How to construct a more efficient ring has been shown in the work on structured overlay networks [SMK+01, SSR08, AS07]. $O(\log n)$ is achieved by skipping entries with exponentially increasing distances. There is also a number of papers showing how to achieve $O(1)$ routing [MA06, RS04, FRGL09]. These topologies are, however, still requiring the basic ring structure as fall-back.

Tree. Tree-like hierarchies are common in computer networks, e.g. DNS [MD88]. For example, a multi-data center topology [AFLV08] starts from a single machine with multiple cpus/cores. Each machine is in a rack connected by a switch and racks are typically grouped in rooms of the data-center. Routing in a tree is $\log p$-hops on average. To achieve efficient routing the tree should be balanced or flat, which either result in a harder to maintain structure or requires more state per tree-node.

We can emulate a multi-tier topology in a single dimensional partitioned name space by using hierarchical prefixes for each tier. The elements in D would then be constructed as $level1.key$, $level2.key$ ... $levelx.key$, where $level1$ corresponds to a data-center, $level2$ to a room and so on. The partition at each level maintain pointers to the partitions in the level below. A change in the partition in one level triggers an update to the parent. It is sufficient if this change eventually propagates to the parent level as long as pointers are not updated with old partitions. Routing starts at the top-level and therefore each level or the leaf partitions should maintain pointers back to the root. Since the assignment of partitions to process groups is flexible, we can ensure locality of levels

and partitions according to some policy. For example, the level *de* ends up on a process group in Germany and *se* in Sweden.

Complete Mapping Table. In a map-based topology each process in the service are maintaining a the complete partition to process group mapping. One way of implementing a decentralized map is by using gossip as in Dynamo [DHJ+07]. Partition owners publishes any partition changes to the gossip network, where newer partitions eventually replaces an existing overlapping or older partition. A route-execution will eventually be redirected to the correct owner by using the local mapping table at each process. In case of a change, the propagation time of the gossip network influences how long time it will take before a route execution finishes. The state overhead is also high since each process must have the complete map in memory for low latency look-ups. This approach has be shown to work for smaller systems, such as a cluster, where the update propagation time is low.

5.3.3 Routecast

To implement the *routecast*-primitive, we use the functions exported by the routing service, *route*, and the process group, *to-multicast*. Based on these abstractions and the partitions state maintained by each process group, the algorithms presented in alg. 11 are straight-forward.

Algorithm 11: The algorithm used to execute *routecast*.

```
1 delivered ← {}
2 procedure routecast(k,msg) do
3 |   route(k, RouteCastRequest(k,msg))
4 on route-receive RouteCastRequest(k,msg) for x
5 |   to-multicast(RouteCastRequest(k,msg))
6 on to-deliver RouteCastRequest(k,msg) do
  |   ▷ Are we responsible for the partition covering k?
7 |   if k ∈ p ∈ partitions and msg ∉ delivered then
8 |   |   rc-deliver(k,msg,p)
9 |   |   delivered ← delivered ∪ {msg}
```

To execute a *routecast* operation, we route a RouteCastRequest(*k*,*msg*) message towards the owner by using the routing service. When *route*-

receive executes for this message a *to-multicast* is invoked with the request (line 4). On the execution of *to-deliver*, the *partitions*-variable is used to check if the process group is still responsible for the partition, and in that case, *rc-deliver* executes with the wrapped message, the key and the partition the message is delivered in.

Correctness. In this section we argue that alg. 11 satisfies property **PTO1-5**. The correctness depends on the properties provided by the TO-multicast implementation, **TO1-5** and the **R1**-property from the routing service. We have not introduced the algorithms for partition management yet, and it is therefore assumed that this executes at a single process group for a single partition. However, alg. 11 also correctly implements **PTO1-5** as we prove in section 5.3.5.

Assume that a process group A has invoked *initialize*($\bot$,$\top$), and thereby storing $([\bot, \top), 1, A)$ in the *partitions*-table. Then, both **PTO1 Partitioned Validity** and **PTO2 Partitioned Integrity** are satisfied by **R1** and **TO1-4**. Any execution of *route*(k, RouteCastRequest(k,*msg*)) eventually results in the execution of *route-receive* at A. As a result, *to-multicast* is invoked in line 5, which in turn results in a *to-deliver* of RouteCastRequest(k,*msg*). For any valid $k \in D$ it is covered by $[\bot, \top)$, corresponding to the partition $p = ([\bot, \top), 1, A)$. Additionally, line 7 and 9 makes sure that *msg* has not been rc-delivered before, guaranteeing at most once delivery. Ensuring at most once delivery using a variable keeping track of all delivered messages is not efficient in terms of memory usage. This can however be improved by using, for example, a time-stamp based approach [GR06].

PTO3 Same Partition Total Order follows from **TO5**. Any pair of messages $(x, m), (y, m')$ are delivered according to a total order for the partition p. The function f in this case is monotonically increasing for each delivered message in A for p. Thus, by **TO5** it is straight-forward to construct f.

Both **PTO4** and **PTO5** follows from **PTO3** since only a single partition exists. We argue for the correctness of these properties after the introduction of the algorithms for partition management.

5.3.4 Partition Management

In this section we describe the algorithms for partition management. In addition, we show that the algorithms holds for the **P1-6**-properties and

analyze their cost.

Partition management includes three operations: *split*, *merge* and *handover*. Both split and merge are trivial operations within a process group when assuming that the group is responsible for the involved partitions. A handover, however, requires coordination between groups since it changes the exclusive ownership of a partition from one process group to another. Since a partition can only be assigned to a single process group at a time (c.f. **P4**), this change must be atomic and eventually terminate. The partition cannot be used until handover termination. If the algorithm never terminates, the partition becomes unusable forever since it is not possible to assume that the algorithm has completed or failed by using timeouts.

A process group is initialized with a table *partitions* for the locally stored partition state, a variable $self$ indicating the current group (*id*, *view*, *members*), and two sets *active* and *pending*. *partitions* have ranges $[a, b)$ as keys and (*start*,*end*,*version*,*owner*)-tuples as values. *start* and *end* represents the range, p_{range}, of a partition p, *version* ($p_{version}$) is the version of the partition and *owner* (p_{owner}) is the partition owner.

Split and Merge

The split and merge algorithms are shown in Algorithm 12. Split and merge are realized as two totally ordered operations using the *to-multicast*-primitive from the process group. Both operations can execute independently in a process group by assuming that a group is responsible for the involved partitions. Any operation executed with a *to-multicast* in a local group is atomic. Therefore, we can perform several modifications to the partitions map within a single invocation of *to-deliver*. A successful split operation must ensure that the previous partition is removed and the two new partitions are added to the map. The merge operation ensures that the merged partitions are removed from the map and their combination is added. Additionally, both operations increases the version number of the new partitions. Thus, for any newer partitioning $\rho \in P_D$, the partitions covering an element in the name space D always has a higher version. Partitions which overlap are causally dependent on each other.

Handover Algorithm

A process group responsible for a partition has exclusive access to perform any modifications to the partition. That is, change the partition

Algorithm 12: Algorithms for splitting and merging partitions.

```
1  procedure Split(x,p) do
2      if x ∈ p_range and x > p_start then
3          to-multicast(Split(x,p))
4  on to-deliver Split(x,p) do
5      p' ← partitions[p_range]
6      if p' and p'_owner = self then
7          k ← p_version + 1                    ▷ Increase the version
8          partitions[[p_start, x)] ← (p_start, x, k, p_owner)
9          partitions[[x, p_end)] ← (x, p_end, k, p_owner)
10         delete partitions[p_range]
11         update-routers((p_start, x, v', self))
12         update-routers((x, p_end, v', self))
13 procedure Merge(a,b) do
14     if a_end = b_start then   ▷ Check that the partitions are adjacent and a < b
15         to-multicast(Merge(a,b))
16 on to-deliver Merge(a,b) do
17     p ← partitions[a_range]
18     q ← partitions[b_range]
19     if p and q and p_owner = q_owner = self then
20         k ← max(p_version, q_version) + 1
21         partitions[[a_start, b_end)] ← (a_start, b_end, k, p_owner)
22         delete partitions[a_range]
23         delete partitions[b_range]
24         update-routers((p_start, x, v', view))
```

range, version or the owner. The idea behind the handover algorithm is to let the current owner update the partition in its local state with the new owner and then telling the new owner that it is responsible. Intuitively, a potential new owner *steals* the partition from the current owner. Since the owner change operation is executed within the process group, the change is atomic. However, there are two complications to the handover protocol. First, a process group may disband at any time as long as it does not store any state and, second, we must handle concurrent handover requests.

Figure 5.3 shows the two phases of the handover protocol presented in algorithm 13. The algorithm uses two sets: the *active* set contains (x,k)-tuples, where $x \in D$ is a name space key and $k \in \mathbb{N}$ is a proposed

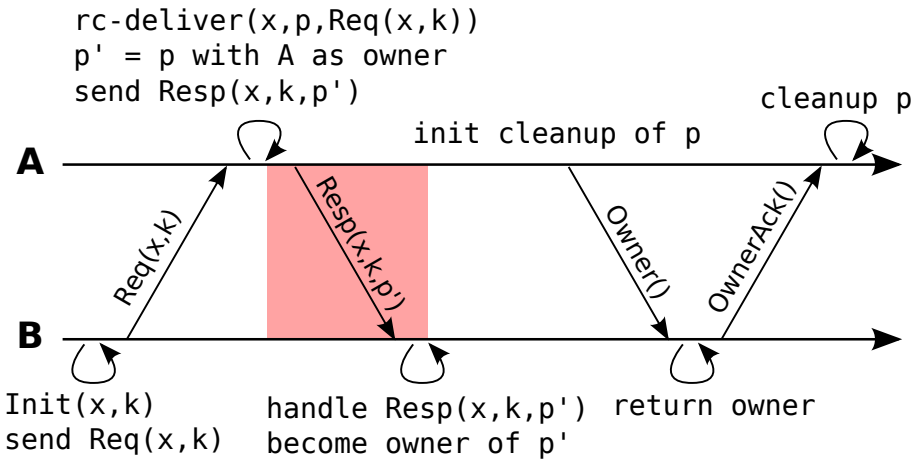


Figure 5.3: Hand-over of a partition p from A to B.

partition version number. This tuple uniquely represents an ongoing handover. *pending* is a set of partition-tuples and indicates all partitions which can be garbage collected after a completed handover.

In the protocol, a process group A tries to retain ownership of a partition p, where $x \in p$. The handover-phase is started from A and the cleanup-phase is initiated from the responsible of x after a successful handover. The handover-phase ensures that the current owner of p, B, gives up the responsibility of p and that A becomes the new owner. The handover request contains $x \in p$ and a proposed new version k for p. Similar to an acceptor in Paxos [Lam01], a process group delivering a valid handover request must always accept the request if k is higher than the current version of p, $p_{version}$. An accepted handover results in an atomic change of owner when executing *rc-deliver* (line 11-14). After this change, the group will not deliver any more messages within the partition. Thus, for concurrent handover requests the first request delivered and accepted "wins" the partition. In the cleanup-phase, B tries to ensure that the handover of p has terminated. That is, the handover-phase has completed when a new process group stores p in the *partitions*-table, and is thereby able to *rc-deliver* messages for p. This may be another group than A, since

A could have received a new handover request in the mean time. When B has completed the clean-up it is free to execute group disband unless it is responsible for other partitions or is part of another handover-operation.

Algorithm 13: Algorithm for handing over a partition p containing x to the initiating process group.

```
1  active ← ∅                                   ▷ Active (not terminated) handovers.
2  pending ← ∅                                  ▷ Pending handover verification
3  procedure handover(x,k) do
4      to-multicast(HandoverInit(x,k))          ▷ Initialize the hand-over with x ∈ D
       and a proposed version k.
5  on to-deliver HandoverInit(x,k) do
6      if (x,k) ∉ active then
7          active ← active ∪ {(x,k)}
8          routecast HandoverRequest(x,k,self) towards x     ▷ Route the
           request towards the group responsible for x.
9  on rc-deliver HandoverRequest(x,k,group) for x ∈ p do
10     if k > p_version then                    ▷ Always accept if the request contains a higher
       proposed version
11         p' ← (p_start, p_end, k, group)
12         pending ← pending ∪ {p'}
13         delete partitions[p_range]
14         send HandoverReply(x,k,p') to group
15     else                                     ▷ k is less than the latest partition version
16         send HandoverReply(x,k,p) to group
17 on receive HandoverReply(x,k,p) do
18     to-multicast(HandoverReply(x,k,p))
19 on to-deliver HandoverReply(x,k,p) do
20     if (x,k) ∈ active and self_id = p_group_id and p_version = k then     ▷ Are we
       responsible for the partition?
21         partitions[p_range] ← p
22     active ← active \ {(x,k)}                ▷ Handover has finished.
```

Avoiding Process Group Disband. Most of the complexity of the protocol is due to the possibility that a process group disbands concurrently with a handover. First, assume a simple RPC-based protocol where B

tries to handover a partition to A by sending a HandoverRequest directly to A and waiting for a HandoverReply. Once the HandoverInit that was invoked to send the HandoverRequest has finished executing at B, a HandoverReply must eventually be received by B in order to terminate the protocol at B. However, if A decides to disband before it has received the HandoverRequest from B, then B will wait indefinitely for a HandoverReply. B can not try to handover the partition to another process group, C, after a time-out since A could be temporarily unavailable or the routed message or the reply can be delayed. Thus, B cannot deterministically terminate the protocol. Initiating the handover from A to a fixed process group B suffers from the same problem. To solve this issue, we modify the simple RPC-protocol with two additions.

First, in order to avoid a disband operation concurrent with a handover, we introduce an *active* set at the initiating process group A and a *pending* set at B (*active* and *pending* in Alg. 13). *active* contains an (x, k)-tuple that uniquely identifies the request, while *pending* contains the partition with the new owner. A process group is free to disband (stop accepting messages) when it is not responsible for any partition and the *active* and *pending* variables are empty.

Second, the HandoverRequest must be sent from A by using a route-request for an element $k \in D$ instead of a direct message from A to B. With a direct message, B could initiate a disband after A has finished the HandoverInit. However, with a routecast-request, the target recipient of HandoverRequest is not bound to a specific process group responsible for k at the time of the send. The handover partner is instead decided upon at the time of execution of *rc-deliver*for the HandoverRequest (line 9). Assuming that a process group B eventually becomes responsible for p, B will *rc-deliver* the HandoverRequest. Since B is not allowed to disband until it has verified that there is a new owner of p, A will eventually receive a HandoverReply by the stubborn resend of the quasi-reliable channel.

The Cleanup Phase. The *handoverCleanup* function from Algorithm 14 is executed after a successful handover at the previous owner, B. It ensures that the *pending*-set is eventually empty, allowing B to disband. Similar to the handover-phase, cleanup starts by sending a *routecast*-request towards the owner of a partition $p \in$ *pending*. Since the time of the handover-execution, the partition could have been modified by a split or

a merge, or another handover could have occurred. However, it is sufficient to make sure that there exists a group responsible for p_{start}. If a group can execute *rc-deliver* for a message routed towards the key p_{start}, either it is p or a newer partition. Any process in B waits for the HandoverUnlockAck and then ensures that all processes removes p from *pending* by execution of a TO-multicast.

Algorithm 14: Clean-up of *handover*-locks from a previous partition owner.

```
1 procedure handoverCleanup(p) do
2 |   routecast HandoverUnlock(p,self) towards p_start      ▷ Only unlock
  |   when there is a responsible for p_start.
3 on rc-deliver HandoverUnlock(p,group) for x ∈ p do
4 |   send HandoverUnlockAck(p) to group
5 on receive HandoverUnlockAck(p) do
6 |   to-multicast(HandoverUnlockAck(p))
7 on to-deliver HandoverUnlockAck(p) do
8 |   pending ← pending \ {p}
```

Partition Availability. The presented algorithm trades availability for consistency. Messages for $k \in p$ can continue to be rc-delivered at an owner B until the HandoverRequest has been delivered. At this point, the ownership of the partition is either changed to A or stays with B depending on the value of k (line 10 in Alg. 13). For a successful handover, in the time between the delivery of the HandoverRequest at B and the *to-deliver* of the HandoverReply message at A, no group can deliver messages for the partition. This means that any message is delayed until the HandoverReply has been to-delivered. In the failure-free case, this time is the latency of a single message send and a to-multicast execution at A.

Abortable Handover

The current algorithm does not allow the process group that *rc-deliver* a HandoverRequest, B, to cancel or abort the handover if the proposed version is higher than the version of the partition. Consider if we would introduce an HandoverAbort(x,k,*partition*) message as a response to an aborted handover. Let the HandoverAbort from B be delayed when

sending back the reply to A. This may lead to a case where A routes a new HandoverRequest. In the mean time, B may have accepted a handover request from C. A's request arrives at C which accepts the handover. In this case we have two messages in-flight, one HandoverAbort and on HandoverReply which would let A to succesfully take responsibility of the partition. If the HandoverAbort arrives first, A terminates the handover since it thinks B is still responsible. However, since C gave up the ownership, no group is responsible for the partition, thereby violating both property **P1 No Gaps** and **P6 Handover Termination**.

Algorithm 15: Algorithm for abortable handover.

```
1  on rc-deliver HandoverRequest(x,k,group) for x ∈ p do
2      if k > p_version then ▷ We may accept since the request contains a higher
       proposed version
3          if abort(x,k,p,group) then
4              p' ← (p_start, p_end, k + 1, self)      ▷ Aborted, increase the version
5          else
6              p' ← (p_start, p_end, k, group)
7              pending ← pending ∪ {p'}
8              delete partitions[p_range]
9          send HandoverReply(x,k,p') to group
10     else                                  ▷ k is less than the latest partition version
11         send HandoverReply(x,k,p) to group
```

Algorithm 15 presents a solution to the *Abortable Handover* problem. We modify the execution of *rc-deliver* for a HandoverRequest to call a function *abort* which returns *true* if it is decided to deny the handover, and *false* otherwise. If the group decides to abort, it proposes a new version of the partition which is larger than the version proposed by the initiating group. This ensures that this handover-request is never accepted by the current group or any other groups if the route-request is re-sent. The main drawback of this approach is that *abort* must be deterministic[2]. It is therefore difficult to base it on, for example, the utilization of the local group without agreeing beforehand on the current usage.

Cost Analysis. The message complexity of a successful handover in the failure-free case depends on 1) the size of the process group and the

[2]The *abort* is executed at each process in a replicated state machine.

algorithm used to implement the to-multicast primitive, and 2) the cost of routing a message in the overlay topology. Let the average group size be g and the average routing steps to be k. First, if the handover succeeds, the cost is $g * 4 + k * 2 + 2$. That is, two to-multicasts per process group and two route requests with replies for the HandoverRequest and the HandoverCleanup. A failed handover results in a cost of $g * 3 + k + 1$ since there is no need to cleanup the *pending* set.

5.3.5 Correctness

The partition management algorithms must ensure six different properties. Split and merge must not violate the **P1 No Gaps** and **P2 No Overlaps** safety properties for a valid partitioning and the operation validity property **P3 Split and Merge Validity**. A handover algorithm must ensure two safety properties, **P4 Exclusive Assignment** and **P5 Handover Validity**, and one liveness property, **P6 Handover Termination**. Furthermore, the partitioned total order properties **PTO4** and **PTO5**, which are necessary for consistent delivery, both relies on the correctness of the partition management.

We prove that algorithms 13 and 14 are not violating the safety properties **P4**, **P5** and that **P6** eventually happens. Intuitively, the correctness of the handover algorithm relies on 1) only the current owner of a partition can change the ownership, 2) the ownership change is atomic, 3) owners must always accept a valid handover request, 4) routing termination and reliable message send between processes in process groups and 5) fault-tolerant process groups ensures eventual termination of the handover and cleanup.

The proof is structured as follows, 1) we infer the validity of split merge (**P3**), 2) we show that both the handover-phase and cleanup-phase eventually terminates at most once at the involved process groups (**P5**) and that either of the groups is the owner (**P6**), 3) we use an induction proof over the history of operations to show that the algorithms cannot violate the exclusive ownership property and that there are no gaps or overlaps in the partitioning (**P1**,**P2**,**P4**). Finally, we use these results when arguing that algorithm 11 correctly implements property **PTO4** and **PTO5**.

Partition Management

Lemma 5.3.1 *Algorithm 12 ensures that a split or merge operation that was initiated by some process eventually terminates and terminates at most once.*

Proof Since both split and merge are depending on TO-multicast, **P3 Split and Merge Validity** can be derived from the **Validity** and **Uniform Integrity** properties of to-multicast. □

Lemma 5.3.2 *A handover (x,k) initiated by a process group A eventually contains $(x,k) \in active_A$ and eventually terminates at most once with $(x,k) \notin active_A$.*

Proof Let A execute to-multicast(HANDOVERINIT(x,k)), then by the **Uniform Agreement** of to-multicast each correct process eventually delivers HANDOVERINIT(x,k). By line 6 and 7 in Alg. 13, $active_A$ contains (x,k) after the delivery if it was not in $active_A$ already, i.e. an ongoing handover for (x,k).

Let $x \in p$ and $B = owner(p)$, then we have two cases, either $k > p_{version}$ or $k \leq p_{version}$ (line 10, 15). If $k > p_{version}$, the handover is accepted and by line 14, B sends a HANDOVERREPLY to A. This reply will eventually arrive by the properties of the quasi-reliable channel and since B is not allowed to disband with p' in $pending_B$ (line 12). B continues to re-send while it still has $p' \in pending$. When A delivers HANDOVERREPLY, by line 19, it will remove (x,k) from $active_A$, thereby terminating the handover at A. If A receives or delivers HANDOVERREPLY more than once, (x,k) is not in $active_A$ and by line 20, the handover cannot terminate successfully again. If $k \leq p_{version}$ and $x \in p' \in pending$ (x is covered by the range of a partition p' in $pending$), B tries to send a HANDOVERREPLY to A by line 16. Even if B disbands before A has received the reply, A will eventually deliver HANDOVERREPLY since it cannot disband until active is empty. It continues to re-send the HANDOVERREQUEST until eventually a process group responds. The handover cannot be accepted by any new owner of p since $k \leq p_{version}$.

□

Lemma 5.3.3 *A successful handover (x,k) initiated by A where $x \in p$ and $p' = (p_{range}, k, A)$ with $p' \in pending_B$ eventually terminates with $p' \notin pending_B$.*

Proof A successful handover of a partition p triggers the execution of $handoverCleanup(p')$ (alg. 14), where p' is the partition after the re-assignment

from a group B to a group A (line 11 in alg. 13). The cleanup verifies that some process group in the system is responsible for p', or a partition derived from p', by executing *routecast*(p_{start}, HandoverUnlock). This message is eventually rc-received by A or another group responsible for p_{start}. By verifying that there is a new process group responsible for p_{start} through the routecast-request, B knows that A has received HandoverReply(x,k,p') and, by line 20 to 22 in alg. 13, was at some point responsible for p'. The eventual delivery of HandoverUnlockAck results in $p' \notin pending_B$ from line 8 in Alg. 14, and thereby terminating the cleanup-phase of the successful handover at B. □

Lemma 5.3.4 *A handover (x, k) terminates at most once.*

Proof Let A deliver HandoverInit for (x, k), from line 6 and 7 in alg. 13, only a single handover for (x, k) can be initiated by A. From lemmas 5.3.2 and 5.3.3, a handover will eventually terminate. The handover terminates when (x, k) is removed from $active_A$ which can happen at most once per handover (line 22). □

Lemma 5.3.5 *A handover (x, k) initiated by a process group A, where $x \in p$ and $owner(p) = B$ terminates with either A as owner of $p' = (p_{range}, k, A)$ or B as owner of p.*

Proof We analyze two cases, when $k \leq p_{version}$ and $k > p_{version}$. It is direct from line 10, 15-16 in alg. 13 that p', with $owner(p') = A$ will never exist when $k \leq p_{version}$. Thus, in this case B is the owner of p when the handover eventually terminates (lemma 5.3.2). When $k > p_{version}$, B cannot be the owner of p by the removal of p from *partitions* at line 13. From this point, it cannot *rc-deliver* any messages for p. Line 11-13 atomically (no concurrent events execute with *rc-deliver* at B) changes the owner of p to A. Furthermore, A eventually becomes the owner of p' by line 14 and line 20-21. Any message loss is handled by the quasi-reliable channel and B is not allowed to disband until *pending* is empty. *pending* can only become empty if A has received the HandoverReply. □

With lemmas 5.3.2, 5.3.3, 5.3.4 and 5.3.5 we show that the algorithms 13 and 14 are correctly solving property **P5 Handover Validity** and **P6 Handover Termination**. **P3 Split and Merge Termination** follows from lemma 5.3.1.

We continue by showing that the changes to the name space does not violate **P1**, **P2** and **P4**. Let a history of partition management operations (*initialize*, *split*, *merge*, *handover*) be denoted H. $A.split$ means that *split* is invoked at a process group A. For example, if there is a partition $p = (a, x, 3, B)$, then a possible history is $H = A.initialize(a, b)$, $A.split(x, (a, b, 1, A))$, $B.handover(a, 3)$. Let H_i be the actual state after i operations in H, where H_0 is the initial partition. Thus, from the example above $p = H_2$. Note that since a split creates two partitions, we have $q, r = H_1 = (a, x, 2, A), (x, b, 2, A)$. In the proofs below we use the version notation for partitions introduced earlier, i.e. p^3 refers to version 3 of p, $(a, x, 3, B)$.

Lemma 5.3.6 *There is no history of operations that violates the properties* ***P1 No Gaps****,* ***P2 No Overlaps*** *and* ***P4 Exclusive Assignment****.*

Proof Lemma 5.3.6 implies that a) no partitions with overlapping ranges nor gaps between partition ranges can be generated by split and merge (**P1** and **P2**) and b) no two partitions with the same range can exist at different owners after a handover (**P4**). We prove this by induction over a history of operations H.

Let the base case be the initial state $H = A.initialize(a, b)$ and $H_0 = (a, b, 1, A)$. A now stores H_0 in the *partitions*-table (alg. 10 line 6). H_0 does not violate any of **P1,2,4** since it is the only partition covering the entire range a, b.

Assume that H_{i-1} is the result of a valid history, then we prove that no operation applied to H_{i-1} leading to H_i violates **P1,2,4**.

Split. Let $H_{i-1} = p = (a', b', k, A)$ be in *partitions* at A. By executing $split(x, p)$, where $a' < x < b'$ we create two new partitions $q = (a', x, k + 1, A)$ and $r = (x, b', k + 1, A)$. These two partitions are not overlapping and there are no gaps between the ranges $[a', x)$, $[x, b')$. This is satisfied by line 8-9 in alg. 12 that creates the new partitions and line 10 which delete the old partition. Furthermore, by line 6 the lines 8-10 are never executed if the partition does not exist. Thus, H_i is a valid after a split.

Merge. Let $H_{i-1} = (p, q)$, where $p = (a', x, k, A)$ and $q = (x, b', k', A)$. By executing $merge(p, q)$, we create a new partition $r = (a', b', k'', A)$, where $k'' = max(k, k') + 1$ to ensure an increased version (causal dependency). The new partition covers the entire range from p and q, thus there are no gaps and no overlaps. Line 21 in alg. 12 creates the new partition and the old partitions are removed at lines 22-23. This code is only exe-

cuted if the two partitions exist and are adjacent (line 14, 19). Thus, H_i is valid after a merge.

Handover. Let $H_{i-1} = p = (a', b', k, A)$, thus, before the handover *partitions* at A contains p, it is stored in. We prove by contradiction. That is, assume that after a handover *partitions* at two different groups contains a partition with the range (a', b'). Let a process group B execute $handover(a', k')$, where $k' > k$ (alg. 13). Assume that HANDOVERREQUEST from B reaches A before any other concurrent handovers and no other concurrent requests changed p. Then, since $k' > k$ (line 10), we execute line 11-14. A new partition $p' = (a', b', k', B)$ is created at line 11 and the old partition is removed from *partitions* at line 13. HANDOVERREPLY eventually arrives at B which then adds p' to *partitions* by line 21. Any change to *partitions* is executed with *to-multicast* which eliminates concurrency. At this point A does not contain p, and B contains p' where $p_{range} = p'_{range}$. This is a contradiction to the earlier assumption that two different groups can contain a partition with the same range. Additionally, there are no gaps and no overlaps.

□

Theorem 5.3.7 *Algorithms 12, 13 and 14 provide a solution for the partition management properties **P1-P6**.*

Proof From lemma 5.3.1-5.3.5, it follows that the properties **P3 Split and Merge Validity**, **P5 Handover Validity** and **P6 Handover Termination**are satisfied. Lemma 5.3.6 shows that no execution history leads to a violation of **P1 No Gaps**, **P2 No Overlaps** and that exactly one process group is responsible for the latest partition, i.e. **P4 Exclusive Assignment**. □

Partitioned Total Order Delivery

In section 5.3.3, we argued for the correctness of the implementation of *routecast* for **PTO1-3** in a single process group since these properties are not affected by the partition management. Here, we extend this argument to include **PTO4 Last Partition Delivery** and **PTO5 Partitioned Total Order**which relies on the correctness of **P0-6**. We start by showing that if a message is delivered for a name space key, there cannot exist another, newer, partition where the message can be delivered (**PTO4**).

Lemma 5.3.8 *(PTO3) For any pair of routed messages (x,m) and (y,m′) that are rc-delivered in a partition p, there exists a total order $\prec_p$ such that if (x,m) $\prec_p$ (y,m′), then rc-deliver(x,m,p) is executed before rc-deliver(y,m′,p).*

Proof We prove that any routed messages (*x*,*m*) that is rc-delivered for a partition p follows a total order $\prec_p$. This follows from line 5 in alg. 11 and property **TO5** for invocations of *to-multicast*, messages within a single process group are delivered according to a total order. Let $f_p : \mathcal{M}_{\mathcal{R}} \rightarrow \mathbb{N}$ be an function from the set of routed messages delivered in p to the totally ordered set $\mathbb{N}$. For each message, f returns the id assigned when performing the *to-multicast* (line 12 in alg. 2). Then $f((x,m)) < f((y,m'))$ if (x,m) was delivered before (y,m'). Thus, there exists a total order $\prec_p$ for *rc-deliver* in the partition p. □

Lemma 5.3.9 *(PTO4) If rc-deliver(x,m,p^k) is invoked at owner(p^k), then there exist no partition $q^{k'}$ where $x \in q^{k'}_{range}$ and $k' > k$.*

Proof We show by contradiction that no operation modifying a partition p^k can have occurred before *rc-deliver*(*x*,*m*,p^k) was invoked. *rc-deliver* is only invoked for a partition if that partition is in the *partitions*-table (line 7-8 alg. 11). All RouteCastRequest-messages are *to-multicast*(line 5), which means there exist a total order on all *rc-deliver* executions. In addition, all partition management operations are ordered in the same process group using *to-multicast*. Thus, there is a total order on all operations accessing the *partitions*-table.

Assume that the *partitions*-table contains p^k such that *rc-deliver*(*x*,*m*,p^k) was invoked for the routed message (*x*,*m*) and that there exist a newer partition $q^{k'}$, where $k' > k$ and x is covered by $q^{k'}$. Then $q^{k'}$ was derived from p^k through one of the partition management operations *split*, *merge* or *handover*. Assume that *partitions* contain p^k at a group A where $owner(p^k) = A$. For *partitions* at A to contain $q^{k'}$, either a *split* or *merge* has executed resulting in $q^{k'}$. However, given lines 10, and 22-23 from alg. 12, the old partition p^k must have been removed. Similarly, for $q^{k'}$ to exists in the *partitions*-table at any other group, the handover-operation for p^k must have executed which according to line 13 in alg. 13 removes p^k from *partitions*. Thus, *rc-deliver*(*x*,*m*,p^k) must have executed before any *split*, *merge* or *handover*-operation resulting in $q^{k'}$. This contradicts that $q^{k'}$ can exist at the same time as *rc-deliver*(*x*,*m*,p^k) was invoked. □

Lemma 5.3.10 *(PTO5) For any pair of routed messages (x,m) and (x,m'), there exist a total order $\prec_x$ for the key x, such that if (x,m) $\prec_x$ (x,m'), then rc-deliver(x,m,p) was invoked before rc-deliver (x,m',q), where both p and q covers x.*

Proof Similar to the proof of lemma 5.3.8, we define a function for the total order $\prec_x$ for messages rc-delivered for a key x, without loss of generality, which is satisfied by alg. 11. There exists a function $g : \mathcal{M}_{\mathcal{R}} \rightarrow \mathbb{N} \times \mathbb{N}$, which given a routed message returns a tuple $\mathbb{N} \times \mathbb{N}$. A pair of tuples returned by g, (a, b) and (a', b') are totally ordered as follows: $(a, b) \prec_g (a', b') \Leftrightarrow a \leq a' \wedge b < b'$. Let g return results based on f from lemma 5.3.8 and a partition p, $g((x, m)) = (p_{version}, f((x, m)))$, where x, m and p is the key, message and partition from *rc-deliver*(x,m,p). $f((x, m))$ from lemma 5.3.8 returns an increasing integer for each rc-delivered message. Furthermore, $\prec_g$ is satisfied as long as the partitions covering x has increasing versions. Since any partition management operation always increases the version number of a partition (lines 7-9, 20-21 in alg. 12 and line 10-11 in alg. 13) and routed messages are always rc-delivered in the latest partition covering a key, this holds true. Thus, by using g there exists a total order $\prec_x$ for any pair of routed messages with x. □

Theorem 5.3.11 *The routecast algorithm 11 together with algorithms 12, 13 and 14 provide a solution for the partitioned total order properties* ***PTO1-PTO5****.*

Proof **PTO1** and **PTO2** follows from the properties **TO1-TO5** and lines 7 and 9 in alg. 11 that ensures at most once delivery of routecasted messages. **PTO3-5** follows from lemma 5.3.8 - 5.3.10, which shows that messages within a partition are delivered according to a total order, messages can only be delivered in the last partition and, finally, any messages delivered to the same key in different partitions are totally ordered. □

5.4 Application State Management

When a partition is handed over to a new owner the application state associated with the partition must still be accessed consistently. The solution for handover presented in section 5.3.3, is only considering the transfer of partition meta-data but not any data stored in the partition. A simple solution to application state transfer would be to include the data in the partition-tuple. This has a major drawback when the state is large. The state size increases the transfer time which in turn influences how long operations to the new owner are delayed. To maintain the consistency, the new owner is not allowed to execute any operations on the partition until the handover is complete.

To reduce this delay, we introduce snapshots and pre-fetching of snapshots. A snapshot is a versioned read-only copy of the application state. Before a group tries to perform a handover, it initiates a snapshot at the current owner. When the snapshot state has been transferred, the pre-fetching phase, the partition handover can execute. At the point of receiving the handover-request at the current owner, it is sufficient to transfer the state that changed since the snapshot. This difference is likely to be significantly smaller than the full state for the partition and thereby reducing the transfer time and delay before the new owner can start handling *rc-deliver* messages. To maintain the reliability guarantees, at least a majority of processes in the new owner group must receive the snapshot data.

The extended handover protocol is presented in alg. 16. We use two new functions: *snapshot* and *diff*. *snapshot* performs a snapshot of the current partition state. This is a callback to the application which should handle snapshots efficiently [SKHH10]. The return is a snapshot partition tuple with (*start*, *end*, *version*, *data*, *snapshot*), where *snapshot* refers to the snapshot-version. This version is used by *diff* to calculate the difference between the current state and a snapshot. In addition, we modify the partition-tuple to contain (*start*, *end*, *version*, *owner*, *data*). The pre-fetching phase returns a snapshot version which is used as a parameter to the modified HandoverRequest (line 1-13). The remaining handover-protocol remains the same except for the *diff*-calculation of the data and the modified partition-tuple.

5.5 Summary and Discussion

This section has presented three modules of the Recode system: the *routing service*, *partition management* and *routecast*. These modules complement the *process group*-module to make the system scalable. Thus, it is possible to add and remove capacity at run-time. The *routecast*-primitive guarantees that operations on state is consistent, even during reconfiguration. This is achieved through a consistent management of partitions as performed by the *partition management*-module. In particular, we have introduced the *handover*-operation which changes the assignment of a partition between two process groups. The handover is efficient and only delays operations in partitions part of a handover with one message send and one process group operation. In the next chapter, we will use the

Algorithm 16: Handover modified to transfer data synchronously.

```
1  procedure handover(x,k) do
2      route PrefetchRequest(x,k,self) toward x        ▷ Route the request
       towards the group responsible for x.
3  on route-receive PrefetchRequest(x,k,group) for x
       ▷ Snapshot the partition covering x.
4      p = findPartitionCovering(parititions, x)
5      p' = snapshot(p)
6      reply PrefetchReply(p',k)
7  on receive PrefetchReply(p',k) do
       ▷ Broadcast snapshot to a majority of group members.
8      send BroadcastSnapshot(p') to members
9      wait until BroadcastSnapshotAck() from ⌈(|members|+1)/2⌉
       ▷ A majority received the snapshot, start handover.
10     x = p'_start
11     if (x,k) ∉ active then
12         active ← active ∪ {(x,k)}
13         routecast HandoverRequest(x,k,p'_snapshot,self) towards x   ▷ Route
           the request towards the group responsible for x.
14 on rc-deliver HandoverRequest(x,k,snapshot_version,group) for x ∈ p do
15     if k > p_version then
16         data = diff(p, snapshot_version)
17         p' ← (p_start, p_end, k, group, data)
18         pending ← pending ∪ {p'}
19         delete partitions[p_range]
20         send HandoverReply(x,k,p') to group
21     else                                  ▷ Unsuccessful handover, k ≤ p_version.
22         send HandoverReply(x,k,p) to group
23 on receive HandoverReply(x,k,p) do
24     to-multicast(HandoverReply(x,k,p))
25 on to-deliver HandoverReply(x,k,p) do
26     if (x,k) ∈ active and self_id = p_group_id and p_version = k then   ▷ Are we
       responsible for the partition?
27         partitions[p_range] ← p
28     active ← active \ {(x,k)}              ▷ Handover finished.
```

routecast-primitive to implement two higher-level services: an array of registers and a lease management service.

Chapter 6

Using Recode

In the previous three chapters, we have introduced the design and implementation of Recode. One of the most important parts of the Recode is the *routecast*-primitive which defines the user semantics for implementing consistent services. *routecast* provides linearizable access to individual keys or partitions in a large name space. Linearizability is the strongest form of consistency and enables single atomic operations on single data objects (keys or partitions). In this chapter we introduce three example applications and their implementation on top of the *routecast*-primitive. The first application is a map with atomic registers, the second is a distributed counter service and the third application implements a lease management service.

Since *routecast* is based on total order broadcast, it has the same guarantees as ZooKeeper [HKJR10] or Chubby [Bur06]. It would therefore be possible to implement the more general coordination service interface they are providing. However, the applications presented below is the implementation from the service side perspective. That is, the code that executes at the processes executing *rc-deliver* after some process has executed *routecast* for some message and key.

6.1 A Map of Atomic Registers

In this section we describe how to implement a map of atomic read/write objects or registers. Each register is referenced using a key and is accessed atomically using the *routecast*-primitive. A single atomic register has the following semantics [GR06]:

Termination Every operation eventually completes

Validity Every read returns the last value written

Ordering If a read returns v_2 after a read that precedes it has returned v_1, then v_1 cannot be written after v_2.

A map with atomic registers is straight-forward to implement with a READ(x) and WRITE(x,v)-operation on top of *routecast*. Algorithm 17 presents the implementation of a read/write-register. The correctness depends on *rc-deliver* which guarantees total order delivery for each element in the name space (**PTO5 Partitioned Total Order**). The efficiency of this implementation depends on the cost of achieving total order since each read/write is ordered through to-multicast. A primary/backup-based implementation can, for example, return a read directly from the primary and the performance is then depending on the read/write ratio. Partitioning of the register map does not have any dependencies at the data level since the resources are independent. An appropriate partitioning is likely to take into account the access frequency of the registers and the size of each value.

Algorithm 17: Read/write register implemented with *routecast*.

```
1 registers ← {}                          ▷ Register id ↦ register value
2 on rc-deliver READ(x) for x ∈ p do
3 |   return registers[x] orElse ⊥
4 on rc-deliver WRITE(x,v) for x ∈ p do
5 |   oldval ← registers[x] orElse ⊥
6 |   registers[x] ← v
7 |   return oldval
```

The execution of *rc-deliver* is atomic at each process in a process group. Therefore, as noted in [ES05], replication using atomic or total order multicast can also provide additional isolation semantics between read and write-operations. For example, arbitrary functions such as compare-and-swap, CAS(x, *oldval*, *newval*), or value incrementation, INCREMENT(x). This abstraction is unnecessary strong for implementing atomic registers which can be done with a quorum approach [ABND95]. However, it exemplifies the simplicity of implementing algorithms requiring coordination with *routecast*. A client that wants to read executes *routecast*(READ(x)) and a writer executes *routecast*(WRITE(x,v)).

6.2 Distributed Counters

A distributed counter service is used to maintain statistics or aggregates for different resources. Such statistics is commonly used at, for example, web-based application that keep track of link clicks, played videos or songs. Each request increments an integer. Note that this cannot be implemented with a read/write register in a single operation (see section 6.1), since the increment first need to read the value to know what to increases, then adds one and finally overwrites the value. To handle this type of operations, read-modify-write operations, we would need to extend the read/write register to somehow handle concurrency. However, with *routecast* the *rc-deliver* execution is atomic for the delivered message and the increment can therefore execute without handling concurrency explicitly. It is already done through the ordering of operations by the TO-multicast implementation.

Algorithm 18: A service for distributed counters.

```
1 counters ← {}                                  ▷ Resource ↦ count
2 on rc-deliver INCREMENT(x) for x ∈ p do
3 |   count ← counters[x] orElse 0
4 |   counters[x] ← count + 1
5 |   return count
6 on rc-deliver COUNTFOR(x) for x ∈ p do
7 |   return counters[x] orElse 0
```

Algorithm 18 presents implementation of a distributed counter service using the *routecast*-primitive. The service increments the counter for a resource k when executing *rc-deliver*(INCREMENT(k)). The increment returns the old value of the counter to the client. To read the value of counter k, the client executes *routecast*(COUNTFOR(k)).

6.3 Lease Management Service

A lease is a time-based lock, that is, it grants exclusive access to some resource for a defined time. A lease management service such as Chubby [Bur06], issues leases to clients that needs to access some resource in the system. A lease-request always returns the current valid lease, this lease

can be held by the client making the access or another client. Internally, the service must first test if there exists a lease and return it, or issue a new lease. This is a read/modify/write-operation (test-and-set) which is strictly stronger than the read/write-operations provided by the atomic register [ES05].

Algorithm 19: A Lease Management Service implemented with *routecast*.

```
1  leases ← {}                          ▷ Lease id ↦ (timestamp, owner)
2  t_lease                              ▷ Time a lease is valid.
3  on rc-deliver GetLease(x, client) for x ∈ p do
4      λ ← leases[x]
5      if λ = ⊥ or λ_timestamp ≥ now() then
           ▷ Invalid or non-existing lease, create a new lease.
6          λ ← (now() + t_lease, client)
7      else if λ_timestamp < now() and λ_owner = client then
           ▷ Lease renewal by the current owner.
8          λ ← (now() + t_lease, client)
9      leases[x] ← λ
10     return λ
```

Algorithm 19 presents the lease management service implementation. It exports a single operation GetLease(*k*,*client*), where *k* identifies the lease and *client* the process trying to acquire the lease. A client executes *routecast*(GetLease(*k*,*client*)), when it tries to become the lease holder. Concurrent clients will be arbitrated depending on the total ordering of the TO-multicast used by *rc-deliver*. A client already holding a lease can renew it by trying to get the lease before the lease time expires.

6.4 Discussion

We have presented three applications built on top of Recode with varying complexity. First, we implemented a map of atomic registers which can be used as a basis for a scalable and consistent key/value-store. Second, we presented a distributed counter service, which addresses a common problem for web-sites keeping aggregate statistics. Finally, we gave an implementation of a lease management service. Leases are often used to guarantee exclusive access to items in storage systems such as files or

blocks. The lease service becomes a bottleneck when serving many small files [MQ09, KHSH]. However, with RECODE it is possible to dynamically add more resources and perform the name space partitioning, which is necessary to scale the application, at run-time.

Although the application logic necessary to implement these applications is rather simple, there are other aspects of making an application scalable. For example, to gain good scalability, the servers must be used fairly. This fairness depends on how the name space is partitioned, something that must be done by the application developer using the *split*, *merge* and *handover*-operations. Achieving a balanced system has been studied extensively in the area of DHTs and can be adapted to RECODE [KR04, DSA07, HK09].

Chapter 7

Evaluation

The handover algorithm is the only operation that requires coordination outside the process group. We analyze the different costs and the fault-tolerance of a handover compared with two other approaches, Risson et al. [Ris07] and Ghodsi [Gho06]. We follow that with two proof-of-concept experiments that evaluates 1) the scalability of the routecast primitive and 2) the run-time reconfiguration of the name space.

7.1 Handover Costs

Table 7.1 compare Risson's protocol [Ris07], FTAR, which uses Fast Paxos Commit with Ghodsi's [Gho06] optimized version of atomic ring maintenance, Scatter [GBKA11] and the handover protocol from Recode. We look at five different costs: 1) the total number of messages, 2) the number of message delays, 3) the number of message delays for which a partition is unavailable for the delivery of routed messages, 4) the number of processes (or process groups) involved in a name space reconfiguration and,

	1)	2)	3)	4)	5)
Recode	$2r_{msg}+2+3g_m$	$2r_{delay}+2+2g_d$	$1+g_d$	2	$2F+1$
FTAR	10	4	4	3	1
Atomic join	$5(1+g_m)$	$5(1+g_d)$	$4(1+g_d)$	3 groups	$2F+1$
Atomic leave	$6(1+g_m)$	$6(1+g_d)$	$6(1+g_d)$	3 groups	$2F+1$
Scatter	$3*g_m+2*g_m$	$4+3*g_d+2*g_d$	$2+2+g_d$	3 groups	$2F+1$

Table 7.1: Cost comparison between the handover protocol, FTAR, Atomic Ring (join/leave) maintenance and Scatter

finally, 5) max processes that can fail. Both handover and Atomic Ring require fault-tolerant processes (RMSs) for correctness in case of failure, we denote the cost of an operation in an RSM g_m and the message delays g_d.

Both phases in the handover protocol include a routing step. The cost of routing varies depending on which routing algorithm and topology is used, we denote the cost r_m for the number of messages and r_d for the message delays. The total number of messages for the handover-phase and the validation-phase is $2r_m + 2$, one routing step and the corresponding reply. Similarly, the message delays are $2r_d + 2$. During a handover, requests to a partition are delayed with one message delay (the reply to the handover request) and one *to-multicast*, before the new owner is responsible and can start answering requests. Finally, there are only two participants involved in the protocol if we ignore the processes executing the route-requests.

Both Atomic Ring and our handover protocol depend on fault-tolerant process groups, which incurs significantly more messages and message delays when compared to FTAR. However, FTAR only allow a single concurrent failure. Atomic Ring and RECODE can handle F failures in a group with $2F + 1$ processes. We also note that in an efficient state machine implementation, for example with primary/backup, only 2 message delays are necessary to execute an operation [CGR07]. Thus, with an efficient implementation a handover in RECODE only requires 3 message delays.

7.2 Implementation and Experiment Setup

We have implemented a proof-of-concept of RECODE using Scala. Each process group member runs in a single JVM and receives messages from the network layer (NIO + Netty) with a single thread. A process group exports the *to-multicast* primitive implemented using the primary/backup protocol described in Chapter 4. New members can be added and existing members removed using the group membership protocol. Primary fail-over uses a lease mechanism as described in [KHSH]. All state is in memory, which makes the system correct as long as there are no power failures affecting a majority of processes in a group[1]. The primary executes one *to-multicast* request at a time and new requests are placed

[1]There were no power outages during the experiments. Durability can be ensured by writing each TO-multicast operation to disk.

in a FIFO queue. The rate of *to-multicast* requests is bounded by the network latency, a majority of group members must ack each multicast request from the primary before the next is executed. Thus, an operation is a single round-trip (2 message delays). We note that there are several techniques to improve the performance of a single process group, such as pipelining and batching of messages [MPP11, CGR07]. However, we keep the group simple to make the analysis of the systems performance easier.

For each of the experiments presented below, we have clients issuing requests synchronously. That is, a client sends out the request and then waits for a reply before sending the next request. If there is no reply within some pre-defined threshold, the request is re-sent and any late reply ignored. We measure the throughput of each client as the completed number of requests per second. The latency is the time from the start of the request until it returns (i.e. the time the client blocks).

We have implemented a simple routing service where each router has a full mapping from partition to process group. On any change: group view change, partition handover, split or merge, the primary in the process group sends an update message to the routing service which broadcasts the update internally to all routers. Clients are routing messages iteratively, that is they send a request for a key and wait for a reply with the process group responsible for the partition covering the key. Requests to a process group member which is not the primary are forwarded without contacting the client. When using the router a client request needs three message delays before reaching the process group responsible for a key. To avoid re-doing lookups for a known key or partition, the client keeps a cache of partitions. An entry is invalidated when the process group no longer is responsible for the partition due to a handover.

The experiments are executed on a cluster with 32 machines connected with a 1Gbps Ethernet switch. The average latency between any two machines is 0.15 ms. We use the 64-bit OpenJDK JVM build 14.0-b16 and Scala 2.9.0-1.

7.2.1 Scalability

Since the name space in a Recode system is partitioned, it is possible to spread the load evenly across both partitions and process groups and thereby machines. With even load over the groups and partition boundaries corresponding to the workload pattern, the system should scale lin-

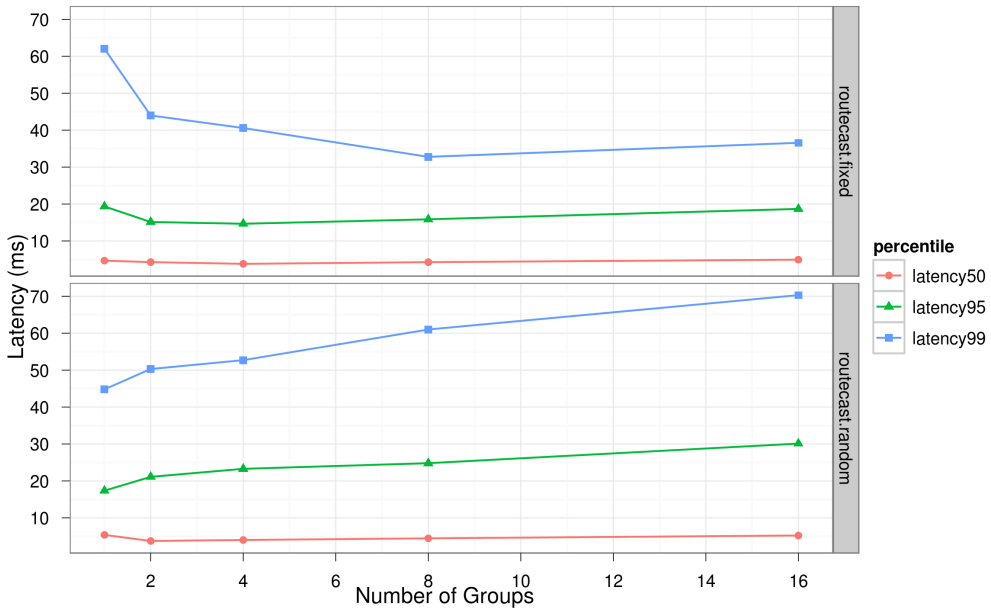
early. In this section we evaluate this claim by comparing our implementation throughput with the expected speedup. The speedup is calculated as $speedup(n) = \frac{tput(n)}{tput(1)}$, where n is the number of groups and $tput(n)$ is the average throughput for n groups.

We have two different client workloads, fixed or randomly generated keys. In the fixed strategy, one or more clients send *routecast*-requests for the same key as fast as possible. Thus, each request always ends up at the same group responsible for the key. At the group a TO-request is generated for each request and it is sufficient with five (because of round-trip time) clients to always keep at least one TO-request in the queue at the primary. Thus, with two groups we need 10 clients and so on. Using a random workload we emulate equally balanced partitions, however, all clients send requests to all process groups unlike in the fixed case.

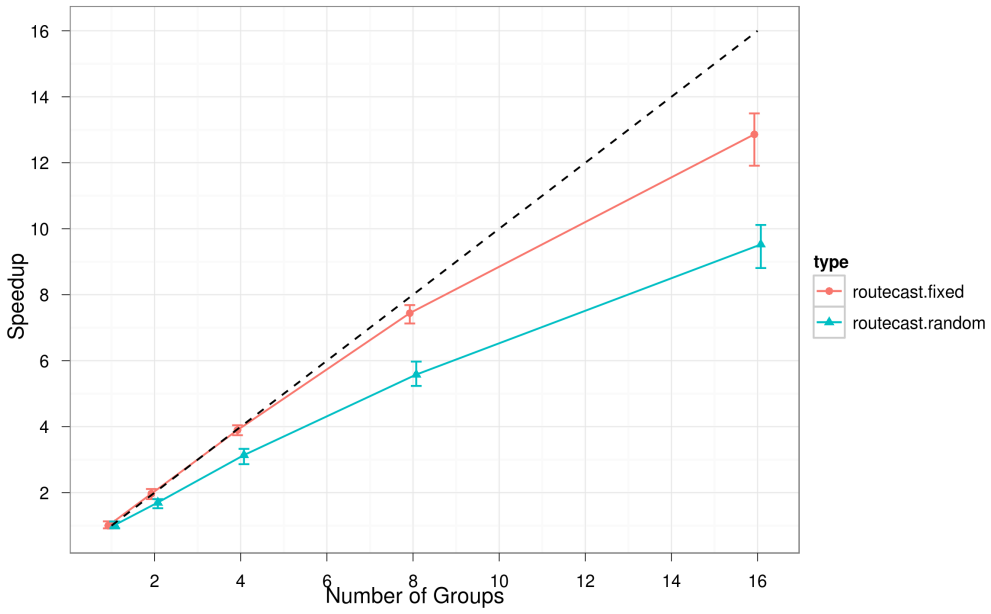
The clients, groups and routers are all executing on different machines. Members of the same group are always on different machines, but the same machine may have several processes. 21 machines are used for hosting group member processes, 5 for clients and 2 for routers. We measure the latency (50th, 95th and 99th percentile) and throughput of requests at the clients.

Figure 7.1 shows the latency and speedup for an increasing number of groups with the different workload strategies. We observe that for the fixed cases, the latency for all percentiles is stable with an increasing number of groups and the throughput increases linearly initially. The overhead in both the latency and speedup comes from the increased number of clients necessary to saturate the queue. The decrease in speedup for 8 and 16 groups is attributed to that multiple processes are using the same physical machine. Each process uses the same network card and sends many small packets. Addtionally, the networking library, netty, has a thread pool for managing connections. With two or more processes per machine we observe a super-linear increase in context switches. The performance reduction is likely to depend on how threads are scheduled or contention between threads when sending and receiving messages, but we have not been able to conclusively trace down the reason. Since the throughput is latency-bound, variations when accessing the underlying hardware have a larger effect on the overall throughput.

For the random cases, we observe a slight increase in latency for the 95th and 99th percentile. Similarly, the speedup decreases faster than for the fixed case. With random requests there is a higher chance that the



(a) Latency



(b) Throughput

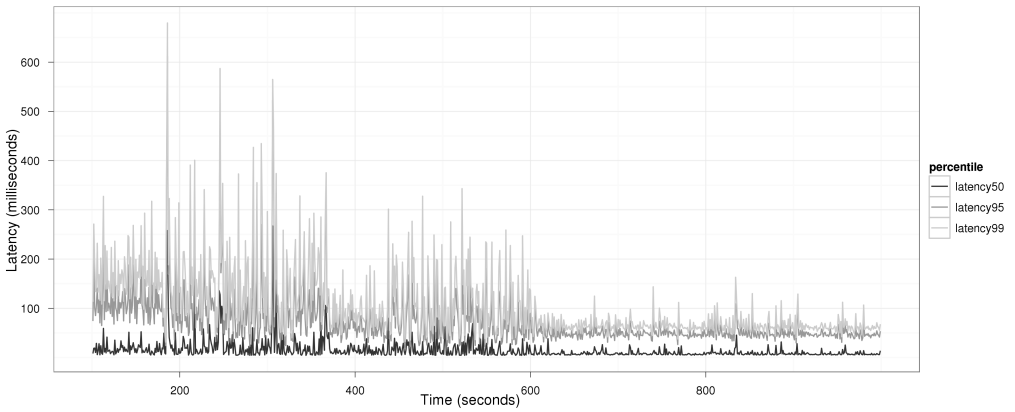
Figure 7.1: Throughput and latency with an increasing number of groups and clients for routecast and multicast.

queues at the primaries are unbalanced. This causes two things, first, the variation of latency increases since requests may stay in a long queue. Second, the probability of an empty queue increases which leads to a sub-linear speedup. An increasing number clients also leads to more variation and does not guarantee that all queues are busy. A solution to this problem is to introduce pipelining, where several requests are executed concurrently. Alternatively, we can create more partitions and groups or ensure that clients only access a fixed set of groups.

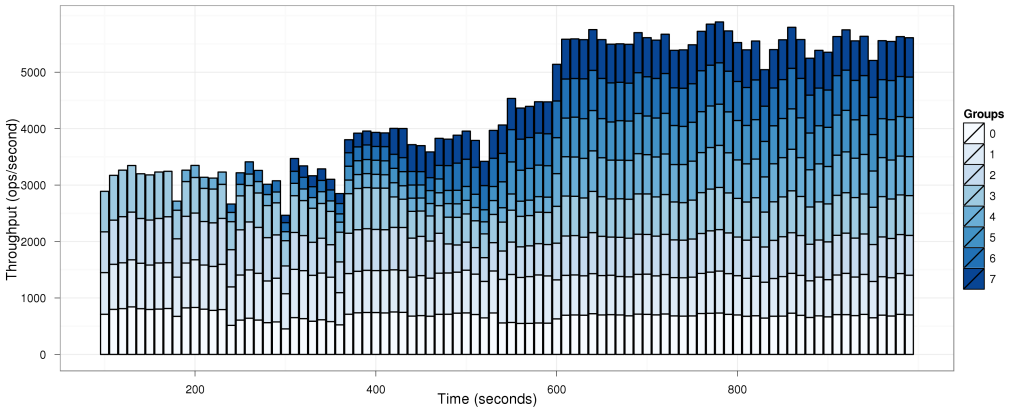
7.2.2 Elasticity

The mechanisms for partition management enables the application to allocate and de-allocate resources at run-time by splitting, merging and moving partitions between groups. We evaluate this mechanism in a single experiment by measuring the throughput and client latencies over time. The system runs with a fixed number of clients (74), groups (4) and partitions (16), after 175 seconds we add 4 more groups and start balancing. A balancing operation uses the handover mechanism to transfer a partition from the group with the most partitions to the group with the least number of partitions. We execute 8 balance operations with a 60 second interval between each operation.

Figure 7.2 contains two plots with the latency and throughput of the system during 10 minutes. From the latency graph in fig. 7.2a, we can see that the system is overloaded initially. With four groups and 74 clients, the queue of TO-multicast operations at each group has a high variance due to the random requests issued by the clients. After the balancing has finished (around 600s), the requests are more evenly distributed and have more queues to choose from which results in reduced latency, variance and higher throughput (c.f. fig. 7.2b). The sudden drop in throughput and increased variance in latency when balancing starts is an effect from the redistribution of requests. Before balancing, each group has four equally sized partitions. The first balance operations places one partition at each new group, this leads to a higher unbalance in the group queues and thereby increased variation and reduced throughput. However, after all 8 balance operations (around 600s) the system quickly stabilizes.



(a) Latency



(b) Throughput

Figure 7.2: Elasticity of the system, new resources are allocated after the vertical line.

7.2.3 Summary

In this chapter we have described a proof-of-concept implementation of RECODE. This implementation has been used to show both scalability and elasticity. When processes do not share the same network card, the system scales linearly. This is however an artifact of the implementation, since keys and groups are both independent (without coordination) the system scales linearly. Additionally, we analytically compared the handover costs with four other approaches. The handover algorithm results in a factor 2 fewer message delays compared to the next-best protocol (Scatter), during which a partition is unavailable.

Chapter 8

Conclusion and Future Work

This thesis has presented Recode, a scalable and fault-tolerant data service with three main properties: full decentralization, consistent data access and reconfigurability. Recode uses concepts from group communication systems to ensure consistency and from partitioned data services and P2P systems to achieve a fully decentralized system. It is composed of four modules: a *process group*, the *routing service*, *routecast* and *partition management*. The specification of these modules makes the system flexible enough to work in environments with different operational requirements. For example, the process group can be optimized for the Wide-Area Network [MJM08, ES07], LAN or cluster [MPP11] or even with byzantine processes [BACdSF08].

Recode addresses the problem of consistent data access in a reconfigurable partitioned name space [Ris07, Gho06, LMR02]. The main issue is how to avoid that two different processes (or replica groups) believe that they are responsible for the same or an overlapping partition. Earlier approaches, based on DHTs, cannot guarantee that exactly one process (or replica group) is responsible for a partition. This is because failure detectors cannot deterministically decide if a process has failed or not [SSM+08]. An incorrect failure detector leads to the revocation of exclusive access to a partition from a process and the assignment of this partition to another process even though the previous owner has not failed. Thus, in those systems a routed message can end up at two different processes and may thereby introduce an inconsistent state.

In a DHT the partition management is tightly coupled with membership management, i.e. a joining, leaving or failing process changes the partitioning. In Recode, process membership is decoupled from partition

management by using process groups. Partition management is done at the level of groups. A consistent reconfiguration of the partitions is done through an atomic handover operation between two groups. Individual processes are therefore never able to revoke or assign partitions based on failure detectors. This makes atomic or consistent data access possible even during run-time reconfiguration of the system.

In addition to the consistent partition management, we introduce the *routecast*-primitive. This primitive guarantees that two messages forwarded towards the same key are delivered at the responsible process group in a total order. This makes it possible to implement linearizability, which is the strongest form of consistency on individual objects[1].

routecast together with the partition management results in a system which is both consistent and reconfigurable at run-time. In Chapter 6, we show three example applications implemented with *routecast*. By making the coordination of operations transparent to the developer, these services can be implemented with relative ease. In Chapter 7, we show that ReCode can be implemented in way that is scalable and which allows for efficient application-level operations during a reconfiguration.

8.1 Future Work

Fault-Tolerance. The process group abstractions are all based on consensus which requires a majority of processes to be correct to make progress. If, however, a majority of processes in a group becomes faulty, the group stops working. This has a major implication for ReCode since if a group responsible for partitions fail, these partitions will become unavailable indefinitely. Is it possible to recover without manual intervention from an administrator if this occurs?

Performance. In [Ped99] Pedone introduced *generic broadcast*. With generic broadcast it is possible to define a pair of messages as conflicting. Two conflicting messages are forced to execute in a total order while if they are not conflicting they can execute concurrently. For ReCode this would mean that when a pair of name space keys in the same partition are not conflicting they can execute in parallel, which should improve on the overall system throughput. However, any reconfiguration operation

[1]Serializability is a stronger form of consistency since it guarantees a global total order for n operations over m data items.

would conflict with operations on the partition. It is unclear if generic broadcast can bring any performance benefits for RECODE.

Acknowledgements

A PhD thesis is a struggle over several years. Nothing is easy, everything is hard. There are, however, several people which have made this struggle easier for me and to whom I'm more than grateful. I'd like to thank Alexander Reinefeld, my Doktorvater, for his supervision that in the end gave me the chance to finish my thesis and for providing me funding through several projects. He also kept believing that I would finish, even though I worked on many different ideas before finding the right topic. Alexander together with Seif Haridi have both constantly pushed me to improve and develop my scientific thinking and writing. They also made it possible for me to spend longer periods with Seif's research group at SICS, Sweden.

I would also like to thank all my colleagues at the computer science research group at ZIB. I shared room with Kathrin Peter during six years, this was a great experience and we had many interesting discussions about both research and life in general. Björn Kolbeck, Jan Stender and Thorsten Schütt always pushed me to be always be critical about my work. Stefan Plantikow was always there to talk about new ideas and allowed me to be open minded. Thomas Röblitz guided me through my first external project, AstroGrid-D, which helped me to survive both SELFMAN and DGSI. Florian Schintke has provided a lot of support with project work. Thomas Steinke, Patrick Schäfer, Nico Kruber, Michael Berlin and others in the group of professor Reinefeld has also been of great support.

SICS was a great experience and I discussed a lot with Cosmin Arad and Tallat Shafaat who worked on the same topic as me. Roberto Roverso gave me insights into doing research in the industry. I would also like to thank my parents who gave me the gift of believing in myself and never giving up. Finally, I thank my wife, Vedrana, for always being there and for her great support and love.

Bibliography

[AAF+94] Afek, Yehuda; Attiya, Hagit; Fekete, Alan; Fischer, Michael J.; Lynch, Nancy A.; Mansour, Yishay; Wang, Da-Wei; Zuck, Lenore D.: Reliable communication over unreliable channels. In: *J. ACM*, volume 41(6):pp. 1267–1297, 1994.

[ABND95] Attiya, Hagit; Bar-Noy, Amotz; Dolev, Danny: Sharing memory robustly in message-passing systems. In: *J. ACM*, volume 42:pp. 124–142, January 1995. ISSN 0004-5411.

[ACT99] Aguilera, Marcos Kawazoe; Chen, Wei; Toueg, Sam: Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. In: *Theor. Comput. Sci.*, volume 220(1):pp. 3–30, 1999.

[ADW10] Adya, Atul; Dunagan, John; Wolman, Alec: Centrifuge: Integrated lease management and partitioning for cloud services. In: *NSDI*, pp. 1–16. USENIX Association, 2010. ISBN 978-931971-73-7.

[AFLV08] Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin: A scalable, commodity data center network architecture. In: Bahl, Victor; Wetherall, David; Savage, Stefan; Stoica, Ion, editors, *SIGCOMM*, pp. 63–74. ACM, 2008. ISBN 978-1-60558-175-0.

[AS07] Aspnes, James; Shah, Gauri: Skip graphs. In: *ACM Transactions on Algorithms*, volume 3(4), 2007.

[ASFPV10] Al-Shishtawy, Ahmad; Fayyaz, Muhammad Asif; Popov, Konstantin; Vlassov, Vladimir: Achieving robust self-management for large-scale distributed applications. In:

SASO, pp. 31–40. IEEE Computer Society, 2010. ISBN 978-0-7695-4232-4.

[AW09] Aguilera, Marcos Kawazoe; Walfish, Michael: No time for asynchrony. In: Fox, Armando, editor, *HotOS*. USENIX Association, 2009.

[BACdSF08] Bessani, Alysson Neves; Alchieri, Eduardo Adílio Pelinson; Correia, Miguel; da Silva Fraga, Joni: Depspace: a byzantine fault-tolerant coordination service. In: Sventek, Joseph S.; Hand, Steven, editors, *EuroSys*, pp. 163–176. ACM, 2008. ISBN 978-1-60558-013-5.

[BDFG03] Boichat, Romain; Dutta, Partha; Frølund, Svend; Guerraoui, Rachid: Deconstructing paxos. In: *SIGACT News*, volume 34(1):pp. 47–67, 2003.

[BGKA10] Beschastnikh, Ivan; Glendenning, Lisa; Krishnamurthy, Arvind; Anderson, Thomas: Harmony: Consistency at scale. Technical Report UW-CSE-10-09-04, University of Washington, September 2010.

[Bir93] Birman, Kenneth P.: The process group approach to reliable distributed computing. In: *Commun. ACM*, volume 36(12):pp. 36–53, 1993.

[BMST93] Budhiraja, Navin; Marzullo, Keith; Schneider, Fred B.; Toueg, Sam: *The primary-backup approach*, pp. 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.

[Bur06] Burrows, Michael: The chubby lock service for loosely-coupled distributed systems. In: *OSDI*, pp. 335–350. USENIX Association, 2006.

[CDG+08] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C.; Wallach, Deborah A.; Burrows, Michael; Chandra, Tushar; Fikes, Andrew; Gruber, Robert E.: Bigtable: A distributed storage system for structured data. In: *ACM Trans. Comput. Syst.*, volume 26(2), 2008.

[CF99] Cristian, Flaviu; Fetzer, Christof: The timed asynchronous distributed system model. In: *IEEE Trans. Parallel Distrib. Syst.*, volume 10(6):pp. 642–657, 1999.

[CGR07] Chandra, Tushar Deepak; Griesemer, Robert; Redstone, Joshua: Paxos made live: an engineering perspective. In: *PODC*, pp. 398–407. ACM, 2007. ISBN 978-1-59593-616-5.

[CHT96] Chandra, Tushar Deepak; Hadzilacos, Vassos; Toueg, Sam: The weakest failure detector for solving consensus. In: *J. ACM*, volume 43(4):pp. 685–722, 1996.

[CKV01] Chockler, Gregory; Keidar, Idit; Vitenberg, Roman: Group communication specifications: a comprehensive study. In: *ACM Comput. Surv.*, volume 33(4):pp. 427–469, 2001.

[CRS+08] Cooper, Brian F.; Ramakrishnan, Raghu; Srivastava, Utkarsh; Silberstein, Adam; Bohannon, Philip; Jacobsen, Hans-Arno; Puz, Nick; Weaver, Daniel; Yerneni, Ramana: Pnuts: Yahoo!'s hosted data serving platform. In: *PVLDB*, volume 1(2):pp. 1277–1288, 2008.

[CT96] Chandra, Tushar Deepak; Toueg, Sam: Unreliable failure detectors for reliable distributed systems. In: *J. ACM*, volume 43(2):pp. 225–267, 1996.

[CZJM10] Curino, Carlo; Zhang, Yang; Jones, Evan P. C.; Madden, Samuel: Schism: a workload-driven approach to database replication and partitioning. In: *PVLDB*, volume 3(1):pp. 48–57, 2010.

[DHJ+07] DeCandia, Giuseppe; Hastorun, Deniz; Jampani, Madan; Kakulapati, Gunavardhan; Lakshman, Avinash; Pilchin, Alex; Sivasubramanian, Swaminathan; Vosshall, Peter; Vogels, Werner: Dynamo: Amazon's highly available key-value store. In: *SOSP*, pp. 205–220. ACM, 2007. ISBN 978-1-59593-591-5.

[DLS88] Dwork, Cynthia; Lynch, Nancy A.; Stockmeyer, Larry J.: Consensus in the presence of partial synchrony. In: *Journal of ACM*, volume 35(2):pp. 288–323, 1988.

[DSA07] Datta, Anwitaman; Schmidt, Roman; Aberer, Karl: Query-load balancing in structured overlays. In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*. 2007.

[DSU04] Défago, Xavier; Schiper, André; Urbán, Péter: Total order broadcast and multicast algorithms: Taxonomy and survey. In: *ACM Comput. Surv.*, volume 36(4):pp. 372–421, 2004.

[ES05] Ekwall, Richard; Schiper, André: Replication: Understanding the advantage of atomic broadcast over quorum systems. In: *J. UCS*, volume 11(5):pp. 703–711, 2005.

[ES07] Ekwall, Richard; Schiper, André: Modeling and validating the performance of atomic broadcast algorithms in high latency networks. In: Kermarrec, Anne-Marie; Bougé, Luc; Priol, Thierry, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pp. 574–586. Springer, 2007. ISBN 978-3-540-74465-8.

[FHIS11] Fetterly, Dennis; Haridasan, Maya; Isard, Michael; Sundararaman, Swaminathan: Tidyfs: a simple and small distributed file system. In: *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pp. 34–34. USENIX Association, Berkeley, CA, USA, 2011.

[FLP85] Fischer, Michael J.; Lynch, Nancy A.; Paterson, Mike: Impossibility of distributed consensus with one faulty process. In: *J. ACM*, volume 32(2):pp. 374–382, 1985.

[FRGL09] Fonseca, Pedro; Rodrigues, Rodrigo; Gupta, Anjali; Liskov, Barbara: Full-information lookups for peer-to-peer overlays. In: *IEEE Trans. Parallel Distrib. Syst.*, volume 20(9):pp. 1339–1351, 2009.

[GBKA11] Glendenning, Lisa; Beschastnikh, Ivan; Krishnamurthy, Arvind; Anderson, Thomas E.: Scalable consistency in scatter. In: Wobber, Ted; Druschel, Peter, editors, *SOSP*, pp. 15–28. ACM, 2011. ISBN 978-1-4503-0977-6.

[GC89] Gray, C.; Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In: *SIGOPS Oper. Syst. Rev.*, volume 23:pp. 202–210, November 1989. ISSN 0163-5980.

[GGL03] Ghemawat, Sanjay; Gobioff, Howard; Leung, Shun-Tak: The google file system. In: *SOSP*, pp. 29–43. ACM, 2003. ISBN 1-58113-757-5.

[Gho06] Ghodsi, Ali: *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, October 2006.

[GL02] Gilbert, Seth; Lynch, Nancy A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *SIGACT News*, volume 33(2):pp. 51–59, 2002.

[GOS98] Guerraoui, R.; Oliveira, R.; Schiper, A.: Stubborn Communication Channels. Technical report, 1998.

[GR06] Guerraoui, Rachid; Rodrigues, Luís: *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540288457.

[HCK+08] Hupfeld, Felix; Cortes, Toni; Kolbeck, Björn; Stender, Jan; Focht, Erich; Hess, Matthias; Malo, Jesus; Martí, Jonathan; Cesario, Eugenio: The xtreemfs architecture - a case for object-based file systems in grids. In: *Concurrency and Computation: Practice and Experience*, volume 20(17):pp. 2049–2060, 2008.

[HHMD05] Haeberlen, Andreas; Hoye, Jeff; Mislove, Alan; Druschel, Peter: Consistent key mapping in structured overlays. Technical Report TR05-456, Computer Scienced Department at Rice University, August 2005.

[HK09] Högqvist, Mikael; Kruber, Nico: Passive/active load balancing with informed node placement in dhts. In: *IWSOS*, volume 5918 of *Lecture Notes in Computer Science*, pp. 101–112. Springer, 2009. ISBN 978-3-642-10864-8.

[HKJR10] Hunt, Patrick; Konar, Mahadev; Junqueira, Flavio P.; Reed, Benjamin: Zookeeper: wait-free coordination for internet-scale systems. In: *Proceedings of the 2010 USENIX ATC Conference*, pp. 11–11. USENIX Association, Berkeley, CA, USA, 2010.

[HW90] Herlihy, Maurice; Wing, Jeannette M.: Linearizability: A correctness condition for concurrent objects. In: *ACM Trans. Program. Lang. Syst.*, volume 12(3):pp. 463–492, 1990.

[JAM10] Jones, Evan P. C.; Abadi, Daniel J.; Madden, Samuel: Low overhead concurrency control for partitioned main memory databases. In: *SIGMOD Conference*, pp. 603–614. ACM, 2010. ISBN 978-1-4503-0032-2.

[JR09] Junqueira, Flavio Paiva; Reed, Benjamin C.: Brief announcement zab: A practical totally ordered broadcast protocol. In: Keidar, Idit, editor, *DISC*, volume 5805 of *Lecture Notes in Computer Science*, pp. 362–363. Springer, 2009. ISBN 978-3-642-04354-3.

[JRS10] Junqueira, Flavio; Reed, Benjamin; Serafini, Marco: Dissecting zab. Technical Report YL-2010-0007, Yahoo! Labs, Jan 2010.

[JRS11] Junqueira, Flavio; Reed, Benjamin; Serafini, Marco: Zab: High-performance broadcast for primary-backup systems. In: *IEEE Int'l Conf. on Dependable Systems and Networks*. June 2011.

[KCSS07] Kannan, Jayanthkumar; Caesar, Matthew; Stoica, Ion; Shenker, Scott: On the consistency of dht-based routing. Technical report, University of California, Berkeley, Jan 2007.

[KHSH] Kolbeck, Björn; Högqvist, Mikael; Stender, Jan; Hupfeld, Felix: Flease - Lease Coordination without a Lock Server. In: *25th IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2011*.

[KLL+97] Karger, David; Lehman, Eric; Leighton, Tom; Levine, Mathew; Lewin, Daniel; Panigrahy, Rina: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: *ACM Symposium on Theory of Computing*, pp. 654–663. May 1997.

[KR04] Karger, David R.; Ruhl, Matthias: Simple efficient load balancing algorithms for peer-to-peer systems. In: *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pp. 131–140. Springer, 2004. ISBN 3-540-24252-X.

[LAB+06] Lorch, Jacob R.; Adya, Atul; Bolosky, William J.; Chaiken, Ronnie; Douceur, John R.; Howell, Jon: The smart way to migrate replicated stateful services. In: Berbers, Yolande; Zwaenepoel, Willy, editors, *EuroSys*, pp. 103–115. ACM, 2006. ISBN 1-59593-322-0.

[Lam79] Lamport, Leslie: How to make a multiprocessor computer that correctly executes multiprocess programs. In: *IEEE Trans. Computers*, volume 28(9):pp. 690–691, 1979.

[Lam96] Lampson, Butler W.: How to build a highly available system using consensus. In: *WDAG*, volume 1151 of *Lecture Notes in Computer Science*, pp. 1–17. Springer, 1996. ISBN 3-540-61769-8.

[Lam98] Lamport, Leslie: The part-time parliament. In: *ACM Trans. Comput. Syst.*, volume 16(2):pp. 133–169, 1998.

[Lam01] Lamport, Leslie: Paxos made simple. In: *ACM SIGACT News*, volume 32(4):pp. 18–25, 2001.

[LJ06] Lampson, Butler; Jackson, Daniel: Principles of computer systems. Technical report, 2006.

[LM10] Lakshman, Avinash; Malik, Prashant: Cassandra: a decentralized structured storage system. In: *Operating Systems Review*, volume 44(2):pp. 35–40, 2010.

[LMR02] Lynch, Nancy A.; Malkhi, Dahlia; Ratajczak, David: Atomic data access in distributed hash tables. In: *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pp. 295–305. Springer, 2002. ISBN 3-540-44179-4.

[LMZ10] Lamport, Leslie; Malkhi, Dahlia; Zhou, Lidong: Reconfiguring a state machine. In: *SIGACT News*, volume 41(1):pp. 63–73, 2010.

[LS05] Ledlie, Jonathan; Seltzer, Margo I.: Distributed, secure load balancing with skew, heterogeneity and churn. In: *INFOCOM*, pp. 1419–1430. IEEE, 2005.

[LT96] Lee, Edward K.; Thekkath, Chandramohan A.: Petal: Distributed virtual disks. In: *ASPLOS*, pp. 84–92. 1996.

[Lyn96] Lynch, Nancy A.: *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

[MA06] Monnerat, Luiz Rodolpho; Amorim, Cláudio L.: D1ht: a distributed one hop hash table. In: *IPDPS*. IEEE, 2006.

[MD88] Mockapetris, Paul V.; Dunlap, Kevin J.: Development of the domain name system. In: *SIGCOMM*, pp. 123–133. 1988.

[Mil91] Mills, David L.: Internet time synchronization: the network time protocol. In: *IEEE Transactions on Communications*, volume 39:pp. 1482–1493, 1991.

[MJM08] Mao, Yanhua; Junqueira, Flavio Paiva; Marzullo, Keith: Mencius: Building efficient replicated state machine for wans. In: Draves, Richard; van Renesse, Robbert, editors, *OSDI*, pp. 369–384. USENIX Association, 2008. ISBN 978-1-931971-65-2.

[MPP11] Marandi, Parisa Jalili; Primi, Marco; Pedone, Fernando: High performance state-machine replication. In: *IEEE Int'l Conf. on Dependable Systems and Networks*. June 2011.

[MPSP10] Marandi, Parisa Jalili; Primi, Marco; Schiper, Nicolas; Pedone, Fernando: Ring paxos: A high-throughput atomic broadcast protocol. In: *DSN*, pp. 527–536. IEEE, 2010. ISBN 978-1-4244-7501-8.

[MQ09] McKusick, Marshall Kirk; Quinlan, Sean: Case study: GFS: Evolution on fast-forward. In: *ACM Queue: Tomorrow's Computing Today*, volume 7(7):p. 10, August 2009. ISSN 1542-7730. doi:http://doi.acm.org/10.1145/1594204.1594206.

[Ped99] Pedone, Fernando: *The Database State Machine and Group Communication Issues*. PhD dissertation, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 1999.

[RD01] Rowstron, Antony I. T.; Druschel, Peter: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pp. 329–350. Springer, 2001. ISBN 3-540-42800-3.

[Ris07] Risson, John: *Reliable Key-Based Routing Topologies*. Ph.D. thesis, The University of New South Wales, 2007.

[RS04] Ramasubramanian, Venugopalan; Sirer, Emin Gün: Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In: *NSDI*, pp. 99–112. USENIX, 2004.

[RSSH09] Reinefeld, Alexander; Schintke, Florian; Schütt, Thorsten; Haridi, Seif: Transactional data store for future internet services. In: *Towards the Future Internet - A European Research Perspective*, 2009.

[RST11] Rao, Jun; Shekita, Eugene J.; Tata, Sandeep: Using paxos to build a scalable, consistent, and highly available datastore. In: *PVLDB*, volume 4(4):pp. 243–254, 2011.

[Sch90] Schneider, Fred B.: Implementing fault-tolerant services using the state machine approach: A tutorial. In: *ACM Comput. Surv.*, volume 22(4):pp. 299–319, 1990.

[Sch06] Schiper, André: Dynamic group communication. In: *Distributed Computing*, volume 18(5):pp. 359–374, 2006.

[Ser10] Serafini, Marco: *Efficient and Low-Cost Fault Tolerance for Web-Scale Systems*. Ph.D. thesis, TU Darmstadt, September 2010.

[SKHH10] Stender, Jan; Kolbeck, Björn; Högqvist, Mikael; Hupfeld, Felix: Babudb: Fast and efficient file system metadata storage. In: *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pp. 51–58. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4025-2.

[SMK+01] Stoica, Ion; Morris, Robert; Karger, David R.; Kaashoek, M. Frans; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM*, pp. 149–160. 2001.

[SSM+08] Shafaat, Tallat M.; Schütt, Thorsten; Moser, Monika; Haridi, Seif; Ghodsi, Ali; Reinefeld, Alexander: Key-based consistency and availability in structured overlay networks. In: *HPDC*, pp. 235–236. ACM, 2008. ISBN 978-1-59593-997-5.

[SSR08] Schütt, Thorsten; Schintke, Florian; Reinefeld, Alexander: Range queries on structured overlay networks. In: *Computer Communications*, volume 31(2):pp. 280–291, 2008.

[ST06] Schiper, André; Toueg, Sam: From set membership to group membership: A separation of concerns. In: *IEEE Trans. Dependable Sec. Comput.*, volume 3(1):pp. 2–12, 2006.

[TTP+95] Terry, Douglas B.; Theimer, Marvin; Petersen, Karin; Demers, Alan J.; Spreitzer, Mike; Hauser, Carl: Managing update conflicts in bayou, a weakly connected replicated storage system. In: *SOSP*, pp. 172–183. 1995.

[WBM+06] Weil, Sage A.; Brandt, Scott A.; Miller, Ethan L.; Long, Darrell D. E.; Maltzahn, Carlos: Ceph: A scalable, high-performance distributed file system. In: *OSDI*, pp. 307–320. USENIX Association, 2006.

[Wie02] Wiesmann, Matthias: *Group Communication and Database Replication: Techniques, Issues and Performance*. PhD dissertation, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2002.

List of Algorithms

Selbständigkeitserklärung

Ich erkläre, dass

- ich die vorliegende Arbeit mit dem Titel: Consistent Key-Based Routing in Decentralized and Reconfigurable Data Services selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist, gemäß Amtl. Mitteilungsblatt Nr. 34/2006.

Berlin, den 8. Februar 2012

Mikael Högqvist